Fundamentals of Computer Science II

05 - Generic - ArrayIntList

Instructor: Varik Hoang

Objectives

- Array List of Integers Class
  - Class Fields
  - Constructors
  - Methods
- Preconditions & Postconditions
- Exceptions
- Generic Data Structure
- Memory Allocation & Garbage Collection

ArrayIntList

Two implementations of the ArrayList class:

- ArrayIntList: A simpler version that only stores ints
- ArrayList<Type>: A generic version

When implementing a new class, two perspectives need to be considered:

- The client’s view - wants to know WHAT the class does (clear specifications), NOT generally interested in implements details
- The implementer’s view - needs to know how to make the object work and how to satisfy the contract

ArrayIntList - Class Fields

```
public class ArrayIntList
{
    // class fields declaration
    public static final int DEFAULT_CAPACITY = 100;
    private int[] elements;
    private int size;
}
```

- Field 1: Since we are implementing ArrayIntList, we need an integer array
- Field 2: An int variable size will keep track of the number of items stored in the array
- Field 3: The array can be initialized using a constant, e.g

ArrayIntList - Constructors - Default Constructor

```java
public class ArrayIntList
{
    // class fields declaration
    ...
    // class default constructor
    public ArrayIntList()
    {
        this.elements = new int[DEFAULT_CAPACITY];
        this.size = 0;
    }
}
```

- Allocate the memory block for the array
- Set the value of the size reference to zero

ArrayIntList - Constructors - Parameterized Constructor

```
public class ArrayIntList
{
    // class fields declaration
    ...
    // class default constructor
    public ArrayIntList()
    {
        this.elements = new int[DEFAULT_CAPACITY];
        this.size = 0;
    }

    // class parameterized constructor
    public ArrayIntList(int capacity)
    {
        this.elements = new int[capacity];
        this.size = 0;
    }
}
```

```
ArrayIntList myList = new ArrayIntList(250);
```

— Create a parameter *capacity* and pass into the constructor

— Use the *capacity* variable instead of DEFAULT_CAPACITY field

ArrayIntList - Constructors - Avoiding Redundancy

```
public class ArrayIntList
{
    // class fields declaration
    ...
    // class default constructor
    public ArrayIntList()
    {
        // call the parameterized constructor
        this(DEFAULT_CAPACITY);
    }

    // class parameterized constructor
    public ArrayIntList(int capacity)
    {
        this.elements = new int[capacity];
        this.size = 0;
    }
}
```

```
ArrayIntList myList = new ArrayIntList(250);
```

ArrayIntList - Methods - Adding Elements

```java
public class ArrayIntList
{
    ...
    // class parameterized constructor
    ...
    // add element to the back of the list
    public void add(int value)
    {
        this.elements[this.size] = value;
        this.size++;
    }

    // add element to specific index of the list
    public void add(int index, int value)
    {
        // shift all elements from the
        // given index to the right by one
        for (int i = this.size; i >= index + 1; i--)
            this.elements[i] = this.elements[i-1];

        // overwrite the value to the given index
        this.elements[index] = value;
        this.size++;
    }
}
```

```java
ArrayIntList myList = new ArrayIntList(250);
myList.add(10);             // the list has [10]
myList.add(0, 20);          // the list has [20, 10]
System.out.println(myList); // [I@5f150435
```

ArrayIntList - Methods - Printing Elements

```
public class ArrayIntList
{
    ...
    // add element to specific index of the list
    ...
    @Override // this method overwrite from Object
    public String toString()
    {
        // base case when list is empty
        if (this.size == 0)
            return "[]";

        // use the fencepost technique
        String result = "[" + this.elements[0];
        for (int i = 1; i < this.size; i++)
            result += ", " + this.elements[i];
        result += "]";
        return result;
    }
}
```

```
ArrayIntList myList = new ArrayIntList(250);
myList.add(10);             // the list has [10]
myList.add(0, 20);          // the list has [20, 10]
System.out.println(myList); // [20, 10]
```

ArrayIntList - Methods - Size & Get/Set

```
public class ArrayIntList
{
    ...
    @Override // this method overwrite from Object
    ...
    // this method return the size of the list
    public int size()
    {
        return this.size;
    }

    // this method get the value at given index
    public int get(int index)
    {
        return this.elements[index];
    }

    // this method set the new value at given index
    public int set(int index, int value)
    {
        return this.elements[index] = value;
    }
}
```

```
ArrayIntList myList = new ArrayIntList(250);
myList.add(10);             // the list has [10]
myList.add(0, 20);          // the list has [20, 10]
System.out.println(myList); // [20, 10]
System.out.println(myList.size()); // 2
System.out.println(myList.get(1)); // 10
myList.set(0, 30);
System.out.println(myList); // [30, 10]
```

ArrayIntList - Methods - Remove/Clear/indexOf

```
public class ArrayIntList
{
    ...
    // this method set the new value at given index
    ...
    // this method remove the element at given index
    public void remove(int index) {
        for (int i = index; i < this.size; i++)
            this.elements[i] = this.elements[i+1];
        this.size--;
    }

    // this method find index of given value in list
    public int indexOf(int value) {
        for (int i = 0; i < this.size; i++)
            if (this.elements[i] == value)
                return i;
        return -1; // not found the item in the list
    }

    // this method clear the list
    public int clear() {
        this.size = 0;
    }
}
```

```
ArrayIntList myList = new ArrayIntList(250);
myList.add(10);              // the list has [10]
myList.add(0, 20);           // the list has [20, 10]
System.out.println(myList); // [20, 10]
System.out.println(myList.size()); // 2
System.out.println(myList.get(1)); // 10
myList.set(0, 30);
System.out.println(myList); // [30, 10]
myList.add(0, 20);  // the list has [20, 30, 10]
System.out.println(myList.indexOf(30)); // 1
myList.remove(1);   // the list has [20, 10]
myList.clear();     // the list has []
```

Preconditions & Postconditions

- The new class and all its method should be **well-documented**
  - **Preconditions** are assumptions that the method makes.
  - **Postconditions** describe what the method accomplishes and assume the preconditions are met.

```
// pre: 0 <= index < size()
// post: returns the integer at the given index in the list
public int get(int index)
{
    return this.elements[index];
}
```

Throwing Exceptions

- We **can not** assume that clients will always satisfy these preconditions.
- **Throw an exception** if a constructor is called with a negative integer.
- Include a string within the parentheses to **make exception more informative**
- Specify the **type of exception** thrown in the method's header's comment

```
public class ArrayIntList
{
    ...
    // class parameterized constructor
    public ArrayIntList(int capacity)
    {
        if (capacity < 0)
             throw new IllegalArgumentException("capacity must not be negative");
        this.elements = new int[capacity];
        this.size = 0;
    }
}
```

Common Exception Types

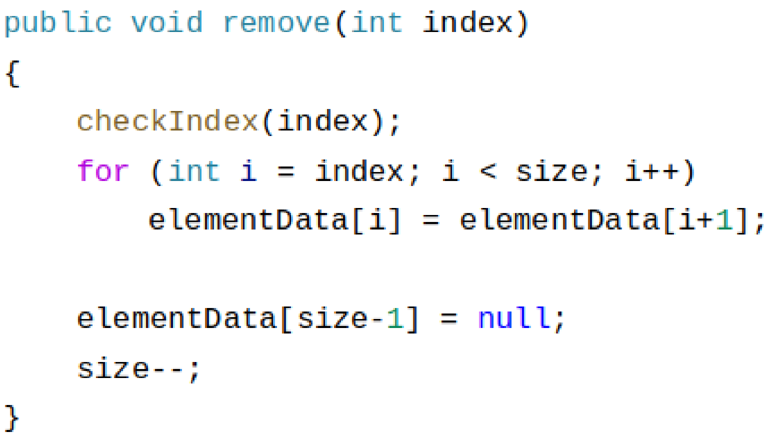
Exception Type	Description
NullPointerException*	A **null** value has been used in a case that requires an object
ArrayIndexOutOfBoundsException*	A value passed as an index to an array is illegal
IndexOutOfBoundsException	A value passed as an index to some nonarray structure is illegal
IllegalStateException	A method has been called at an illegal or inappropriate time
IllegalArgumentException	A value passed as an argument to a method is illegal
NoSuchElementException	A call was made on an iterator's next method when there were no values left to iterate over

Geneic Data Structure

```
public class ArrayList<Type>
{
    // class fields declaration
    ...
    private Type[] elements;
    ...
    // class parameterized constructor
    public ArrayList<Type>(int capacity)
    {
        if (capacity < 0)
             throw new IllegalArgumentException("capacity must not be negative");
        this.elements = (Type[]) new Object[capacity];
        this.size = 0;
    }
    ...
    // this method find index of given value in list
    public int indexOf(Type value)
    {
        for (int i = 0; i < this.size; i++)
            if (this.elements[i].equals(value))
                return i;
        return -1; // not found the item in the list
    }
}
```

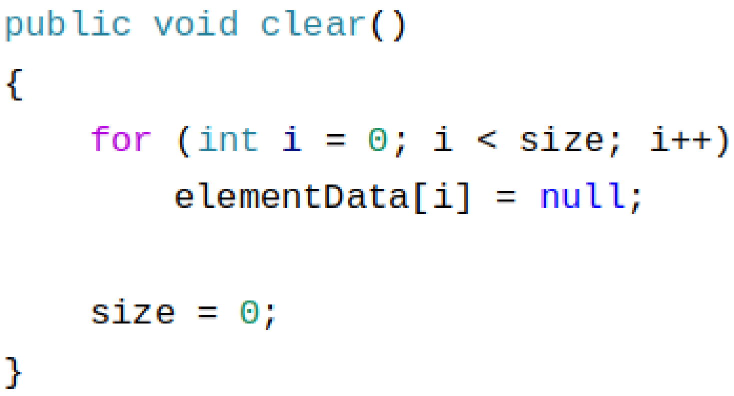
Memory Allocation & Garbage Collection

- We should remove object is no longer needed.
- The garbage collector looks for objects that are no longer being used.
- the garbage collector won't "recognize" if the **ArrayList** is still keeping references to these elements.
- To "free up" an object, values should be set back to null.



```
public void remove(int index)
{
    checkIndex(index);
    for (int i = index; i < size; i++)
        elementData[i] = elementData[i+1];

    elementData[size-1] = null;
    size--;
}
```



```
public void clear()
{
    for (int i = 0; i < size; i++)
        elementData[i] = null;

    size = 0;
}
```

Questions & Answer

Thank You!