Chapter 17 | ArrayLists

Java AP

Copyright Notice
Copyright © 2013 DigiPen (USA) Corp. and its owners. All Rights Reserved
No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052
Trademarks
DigiPen® is a registered trademark of DigiPen (USA) Corp.
All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged.

ArrayLists

- ArrayList is a class in Java that allows the creation of arrays with dynamic (not fixed) numbers of elements.
- ArrayLists have a **size**, which is the current number of elements in the list.
- ArrayLists also have a **capacity,** which is the current maximum number of elements. This value may be less than the ArrayList's size.
- An ArrayList's capacity increases automatically when the number of elements goes over the current capacity.
- Instances of the ArrayList class can store objects, but not primitive types. In order to use primitive types with ArrayLists, we need to use something called a **wrapper class**.

Wrapper Classes

- A wrapper class is a class that takes an existing type, either a primitive type or a class, and wraps it up in a new class with a different interface.
- Java has wrapper classes for all of its primitive types. These wrappers make it possible to use them with ArrayLists and other classes that can only work with object types.
- The wrapper class for integers is called *Integer.* Similarly, the wrapper class for double is *Double,* the wrapper for float is *Float*, and so on.
- Initializing a wrapper object can be done in a number of ways. The results of the following statements are equivalent:
    - `Integer wrapper = new Integer(4);`
    - `Integer wrapper = new Integer.valueOf(4);`
    - `Integer wrapper = new Integer("4");`
    - `Integer wrapper = 4;`
- **Note:** The last statement will not work in versions of Java earlier than 5.0.

Example	
	```
	public class Main
	{
	      public static void main(String[] args)
	      {
	            Integer wrapper = new Integer(4);
	            System.out.println(wrapper);
	      }
	}
	```

Output	4

© 2013, DigiPen Institute of Technology. All Rights Reserved

Explanation	We create an instance of the integer wrapper class **Integer** by using new and assign it to a variable. The integer that is passed in to the constructor is stored in a class field that has a type of *int*. Since Integer is a class and not a primitive type, passing the wrapper instance to the print method calls the class's toString method to create a string representation of the object. The toString method will return the string representation of the integer we passed in when we created the instance, which is "4". This value is then printed to the console.

Autoboxing and Unboxing

- Java is often smart enough to convert from a primitive type to a wrapper object when necessary (and vice versa).
- When Java automatically converts from a primtive to a wrapper, it is called **autoboxing.** When Java converts from a wrapper to a primitive, it is called **unboxing.**
- Although you may not have realized it, we've already seen an example of autoboxing:
    - `Integer wrapper = 4;`

- In the above statement, the instance of the Integer class, wrapper, and the integer literal, 4, are of two different types. However, Java recognizes that 4 can be converted to an Integer wrapper object and "autoboxes" the integer (does the conversion for us) before performing the assignment.
- Autoboxing occurs in the following situations:
    - A primitive type is assigned to an instance of the corresponding wrapper class.
    - A primitive type is passed to a method that takes the corresponding wrapper class.
- The reverse process, unboxing occurs in the following situations:
    - A wrapper class is assigned to a variable of the corresponding primitive type.
    - A wrapper class is passed to a method that takes the corresponding primitive type.

Example

```
public class Main
{
	public static void main(String[] args)
	{
		Double one = 1.0;
		Double two = 2.0;
		Double half = one / two;
		System.out.println(half);
	}
}
```

Output	`0.5`
Explanation	Two instances of the Double wrapper class are created. Since 1.0 and 2.0 are double primitives, Java performs autoboxing to assign them to the two Double instances. Since the division operator only works for primitive types, performing division with Double wrappers forces Java to unbox them before it can evaluate the result. The result, which is also a primitive, is then assigned to a new Double wrapper, which requires Java to use autoboxing one last time.

Explicit unboxing

- It is also possible to manually unbox a wrapper object using methods built into each of the wrapper classes for Java's primitive types. This is known as *explicit* unboxing, which contrasts Java's automatic or *implicit* unboxing.
- To explicitly unbox a wrapper object as an integer, we would use the **intValue()** method. Similarly, to unbox as a double, we would use **doubleValue()**.

Example	
	```
	public class Main
	{
	      public static void main(String[] args)
	      {
	           Integer wrapper = new Integer(4);
	           System.out.println(wrapper.intValue());
	      }
	}
	```

Output	`4`
Explanation	An instance of the Integer class is created and given a value of 4. In the print statement, instead of letting the wrapper return its value as a string (which would be done implicitly when passed to the print method), we explicitly tell the wrapper to return its value as an int, which is then passed to the print function. **Note:** In most cases there is no advantage to explicitly unboxing a variable. If your code works without explicit unboxing, it's often best to leave it that way.

© 2013, DigiPen Institute of Technology. All Rights Reserved

Changing the value of a wrapper object

- Similar to Strings, all of Java's wrapper classes are immutable, meaning instances of the classes cannot be directly altered after they have been created.
- The only way to change the value of a wrapper object is by creating a new object and assigning it to the wrapper variable.

Example	
	```
	public class Main
	{
		public static void main(String[] args)
		{
			Integer wrapper = new Integer(4);
			System.out.println(wrapper);
			wrapper = 10;
			System.out.println(wrapper);
		}
	}
	```

Output	
	```
	4
	10
	```

Explanation	An instance of the Integer class is created and given a value of 4. In the print statement, instead of letting the wrapper return its value as a string (which would be done implicitly when passed to the print method), we explicitly tell the wrapper to return its value as an int, which is then passed to the print function. **Note:** In most cases there is no advantage to explicitly unboxing a variable. If your code works without explicit unboxing, it's often best to leave it that way.

Creating ArrayLists

- To use ArrayLists, you will first need to import **java.util.ArrayList** or, alternatively, import everything in the java.util package by importing **java.util.***.
- Once you have the correct package imported, you can declare and initialize an ArrayList using the following syntax:
    - `ArrayList<Integer> intList = new ArrayList<Integer>();`
    - `ArrayList<String> strList = new ArrayList<String>();`

- **Note:** Only Integer and String are shown here, but the syntax is identical for all other object types.
- As stated previously, we can only create ArrayLists of objects. Attempting to create an ArrayList that uses one of the primitive types will result in an error.

Example	
	```
	public class Main
	{
	  public static void main(String[] args)
	  {
	    ArrayList<Integer> list1 = new ArrayList<Integer>();
	    ArrayList<int> list2 = new ArrayList<int>();
	  }
	}
	```

Output	
	```
	Exception in thread "main" java.lang.Error: Unresolved compilation
	problems:
		Syntax error on token "int", Dimensions expected after this token
		Syntax error on token "int", Dimensions expected after this token

		at edu.digipen.apcs.Main.main(Main.java:7)
	```

Explanation	The first statement of the main method, initializing an ArrayList with the Integer type, is valid, but the second is not. The error message is a bit obtuse, but essentially what's happening is that the compiler is getting confused because we tried to specify a primitive type for the element type of our ArrayList. This is due to the fact that ArrayLists use something called **generics,** which makes it possible for types to be used as parameters when creating instances of a class. When a class requires a type as one of its parameters, that type is specified separately from other parameters and is enclosed by angle brackets (Ex: <Integer>). Generics allow programmers to create classes that are much more flexible, with the caveat that only *object types* can be used as parameters.

Adding elements during initialization

- Data can be added to an ArrayList during initialization by using the **Arrays.asList()** function.
- The Arrays.asList() function can take a variable number of comma-separated elements. These elements are then combined into a list that we can use as data for initializing the ArrayList.
- Example:

  -
    ```
    ArrayList<Integer> arrayList = new
    ArrayList<Integer>(Arrays.asList(1, 2));
    ```

- Being able to add items to our list doesn't do us much good if we can't access them, though. We need to know how to get and set individual elements as well as the proper way to traverse the list.

Traversing ArrayLists

- Once an ArrayList has been created, it can be traversed using loops much like a normal array. ArrayLists can be traversed with for loops as well as for-each loops.
- Since ArrayLists are not arrays, we cannot use the bracket operators [ ] to access individual elements in the list. However, ArrayLists have accessors and mutators, **get()** and **set()**, which provide the same functionality.
- You may recall that arrays have a length property that tells us the number of elements in the array. Similarly, we can find the number of elements in an *ArrayList* by using the **size()** function.

Example 1

```
import java.util.ArrayList;
import java.util.Arrays;
public class Main
{
  public static void main(String[] args)
  {
    ArrayList<Integer> arrayList =
      new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));

    for(int i = 0; i < arrayList.size(); ++i)
    {
      System.out.println(arrayList.get(i));
    }
  }
}
```

Output

```
1
2
3
```

	4 5
Explanation	An ArrayList is initialized with the elements 1, 2, 3, 4, 5. Then a for loop is used to print out each element in the list. The **size()** function is used to determine the upper bound for the counter variable, i, in the for loop. Since the ArrayList is not an array, the **get()** function must be used to access the elments in order to print them out. The get function takes a single argument, the index of the element we want to access, and returns that element if the index is valid.

Example 2	
	```
	import java.util.ArrayList;
	import java.util.Arrays;
	public class Main
	{
	  public static void main(String[] args)
	  {
	    ArrayList<Integer> arrayList =
	      new ArrayList<Integer>(Arrays.asList(1,2,3));

	    for(int i = 0; i <= arrayList.size(); ++i)
	    {
	      System.out.println(arrayList.get(i));
	    }
	  }
	}
	```

Output	
	```
	1
	2
	3
	Exception in thread "main"
	java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
	     at java.util.ArrayList.rangeCheck(Unknown Source)
	     at java.util.ArrayList.get(Unknown Source)
	     at edu.digipen.apcs.Main.main(Main.java:11)
	```

Explanation	An ArrayList is initialized with the elements 1, 2, 3. Then a for loop is used to print out each element in the list.

	This time, we changed our for loop so that it stops when our counter is less than or equal to size. As a result, the counter goes from 0 to 3, but since an array of size 3 only goes from 0 to 2, the program crashes. As with normal arrays, trying to access an element using an invalid index, one that is outside the bounds of the list, will result in an *IndexOutOfBoundsException.*

Example 3	
	```
	import java.util.ArrayList;
	import java.util.Arrays;
	public class Main
	{
	  public static void main(String[] args)
	  {
	    ArrayList<Integer> arrayList =
	      new ArrayList<Integer>(Arrays.asList(1,2,3,4,5));

	    for(Integer num : arrayList)
	    {
	      System.out.println(num);
	    }
	  }
	}
	```

Output	
	```
	1
	2
	3
	4
	5
	```

Explanation	An ArrayList is initialized with the elements 1, 2, 3, 4, 5. Then a for-each loop is used to print out each element in the list. Notice how using a for-each loop in this instance completely alleviates the need for the **size()** and **get()** functions. When write access is not necessary, for-each loops allow for much more elegant code.

Inserting Elements

- There are many different ways to add individual elements to an ArrayList after it has been created.
- To insert an element into the list, we can use the **add()** function. This function requires one argument, the *element* to be added to the list. If no other information is given, this function will append the given element to the end of the list.
- The add function also has another version in which the first argument is instead the *index* at which to add an element. The *element* to be added is given as the second argument.
- **Note:** The type of the element passed to the add function must match the type specified in angle brackets < > when the ArrayList was created.

Example 1

```java
import java.util.ArrayList;
import java.util.Arrays;
public class Main
{
  public static void main(String[] args)
  {
    ArrayList<Integer> arrayList =
      new ArrayList<Integer>(Arrays.asList(1,2));

    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();

    arrayList.add(3);
    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();
  }
}
```

Output

```
1 2
1 2 3
```

Explanation

An ArrayList is initialized with the elements 1, 2. Then an element is added to the list using **add().** Since no index is specified, the element is appended at the end of the list. Notice how the second time the list is printed, it correctly shows that it has one additional element.

ArrayLists let us add and remove elements from a list without having to constrain the list to a specific fixed size. This is useful since it is often difficult to predict exactly how many elements a list will need to store.

Example 2

```java
import java.util.ArrayList;
import java.util.Arrays;
public class Main
{
  public static void main(String[] args)
  {
    ArrayList<Integer> arrayList =
      new ArrayList<Integer>(Arrays.asList(1,2,4,5));

    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();

    arrayList.add(2, 3);
    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();
  }
}
```

Output

```
1 2 4 5
1 2 3 4 5
```

Explanation

An ArrayList is initialized with the elements 1, 2, 4, 5. Then an element is added to the list using the other version of **add().** 2 is specified as the index at which to insert the element, and the element to be inserted is 3. Index 2 is currently occupied by 4, so the portion of the list starting at index 2 is shifted over one place and the value at index 2 becomes 3.

Note: If we had tried to add an element at index 5, which is greater than the size of the ArrayList, the program would have crashed. Passing an index that is (a) less than 0 or (b) greater than or equal to size will always result in an *IndexOutOfBoundsException.*

Changing the value of an element

- As mentioned earlier, ArrayLists have both accessors and mutators. We have already covered the get() method, but we have not yet seen the **set()** method.
- The ArrayList's **set()** method can be used to replace an existing element with a new one. Just like the second version of the add method, the set method takes two arguments, the *index* of the value to be replaced, and the *element* with which to replace it.

Example	
	```
	import java.util.ArrayList;
	import java.util.Arrays;
	public class Main
	{
	  public static void main(String[] args)
	  {
	    ArrayList<Integer> arrayList =
	      new ArrayList<Integer>(Arrays.asList(1,2));

	    for(Integer num : arrayList)
	    {
	      System.out.print(num + " ");
	    }
	    System.out.println();

	    arrayList.set(1, 3);
	    for(Integer num : arrayList)
	    {
	      System.out.print(num + " ");
	    }
	    System.out.println();
	  }
	}
	```

Output	
	```
	1 2
	1 3
	```

Explanation	An ArrayList is initialized with the elements 1, 2. Then the element at index 1, the second element, is replaced with an Integer with the value 3. Calling the **set()** method does not change the number of elements in the list. Both for-each loops print the same number of elements, but by the second for-each loop, the second element of the list has been replaced.

Deleting Elements

- ArrayList also has a few different methods that we can use to dynamically remove elements from the list. We can remove elements at a given index or we can remove elements that match a specific object of the same type.
- The method used to accomplish both of these tasks is called **remove().** Like the add method, the remove method has two different versions.

Example 1

```java
import java.util.ArrayList;
import java.util.Arrays;
public class Main
{
  public static void main(String[] args)
  {
    ArrayList<Integer> arrayList =
      new ArrayList<Integer>(Arrays.asList(1,2));

    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();

    arrayList.remove(1);
    for(Integer num : arrayList)
    {
      System.out.print(num + " ");
    }
    System.out.println();
  }
}
```

Output

```
1 2
1
```

Explanation

An ArrayList is initialized with the elements 1, 2. Then the element at index 1, the second element, is removed.

This version of the remove method has the following signature:

```
    E ArrayList.remove(int index)
```

E is the element that was removed and *index* is the index of the element that was removed. If you need to keep the element you removed around for later use, you can store the value returned by the method in a variable.

	Note: As with other similar ArrayList methods, attempting to access an element using an index less than 0 or beyond the size of the list will result in an *IndexOutOfBoundsException*.

Example 2

```java
import java.util.ArrayList;
import java.util.Arrays;
public class Main
{
  public static void main(String[] args)
  {
    ArrayList<Integer> arrayList =
      new ArrayList<Integer>(Arrays.asList(1,2));

    boolean found = arrayList.remove(new Integer(1));
    if(found)
    {
      System.out.println("1 was found and removed.");
    }
    else
    {
      System.out.println("1 was not found.");
    }

    found = arrayList.remove(new Integer(3));
    if(found)
    {
      System.out.println("3 was found and removed.");
    }
    else
    {
      System.out.println("3 was not found.");
    }
  }
}
```

Output

```
1 was found and removed.
3 was not found.
```

Explanation

An ArrayList is initialized with the elements 1, 2. Then the first element matching the number 1 is removed, as well as the first element matching the number 3. Because there was no element in the list matching the number 3, the second remove operation has no effect.

© 2013, DigiPen Institute of Technology. All Rights Reserved

The version of remove used here is significantly different than the one in the previous example. It has the following signature:

```
bool ArrayList.remove(E element)
```

Instead of taking the index of an element to remove, this method takes an object of the same type as the elements in the list. The method looks for the first object that matches this object and removes it. It then returns true if it found and removed an element, or false if it did not find any matches.

Important: In order to use this version of remove with an ArrayList containing instances of the Integer class, you must supply the function with an Integer object, otherwise the other remove method will be called. This may seem counterintuitive, but let's take a look at the signatures of both methods, this time with the generic type E replaced with Integer:

```
Integer ArrayList.remove(int index)
bool ArrayList.remove(Integer element)
```

When calling an overloaded method and passing it a variable of type *int*, Java will pick the method that best matches the provided name and arguments. Since the argument we're passing has type *int,* it's an exact match with the first remove method. In order to call the second remove method, we have to provide something that matches the second method better than the first, so we will instead use an Integer object.

Note: When comparing a list element to the given object, this version of remove will use the class's **equals()** method. If the class does not have the equals method overridden, it will end up comparing the two objects' memory addresses, not their values.