Chapter 19 | Lists

Java AP

The List Interface

- The **List interface** is an interface provided by Java that other classes can implement to create a class that represents a list of objects. Classes that implement the List interface provide methods that are useful for working with collections of similar objects.
- We've already used one class that implements the List interface: ArrayList. Two other commonly used List classes are **LinkedList** and **Vector**.
- Listed below are the methods in the List interface that we've already seen. (Note: E refers to the type of Object we are storing in the list.)

Method Signature	Description
boolean add(E e)	Adds an element to the end of the list.
void add(int index, E element)	Adds an element at the given index, moves all elements at or greater to the index one to the right.
E get(int index)	Returns the element at the given index.
int indexOf(Object o)	Returns the index of the specified object if it is in the list, -1 if it is not.
E remove(int index)	Removes the element at the given index, returning a reference to the removed object.
boolean remove(Object o)	Removes the first instance of the specified object from the list. Returns true if it was found and removed, false if it was not found.
E set(int index, E element)	Replaces the element that currently resides at the given index with the provided element.
int size()	Returns the size of the List (the number of items currently stored in it).

- A full listing of the methods in the List interface can be found at the link below. Many of them will be demonstrated throughout the course of this chapter.
  http://docs.oracle.com/javase/7/docs/api/java/util/List.html#method_summary

Creating a List

- Since List is an interface, it cannot be instantiated directly.
- However, a List reference can be used to refer to any class that implements it.

Example	`import java.util.*;` `public class Main` `{` `  public static void main(String[] args)` `  {` `  List<Integer> arrayList = new ArrayList<Integer>(` `  Arrays.asList(1,2,3,4,5));` `  List<Integer> linkedList = new LinkedList<Integer>(` `  Arrays.asList(1,2,3,4,5));` `  List<Integer> vector = new Vector<Integer>(` `  Arrays.asList(1,2,3,4,5));` `  }` `}`
Explanation	ArrayList, LinkedList, and Vector all implement the List interface. Thus, we can assign objects of their types to List references, provided the specified type of the objects being stored matches. Typing `List<Float> arrayList = new ArrayList<Integer>()`, for example, would result in a compilation error.

Printing a List

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

	`System.out.println("ArrayList: " + arrayList);` `System.out.println("LinkedList: " + linkedList);` `System.out.println("Vector: " + vector);` `}` `}`
Output	`ArrayList: [1, 2, 3, 4, 5]` `LinkedList: [1, 2, 3, 4, 5]` `Vector: [1, 2, 3, 4, 5]`
Explanation	Since each List class implements the **toString()** method, we can print List objects just as we would any other variable. The String returned by the List will contain all the elements in the list.

Finding elements in a List

- It is often useful to know whether a list contains a certain element or elements. The List interface has two methods for providing this functionality.

Method Signature	Description
boolean contains(Object o)	Returns true if the list contains an element that matches the given object, false otherwise.
boolean containsAll(Collection<?> c)	Returns true if the list contains all the elements in the given collection, false otherwise.

- **Note**: The "Collection" type referred to above is another interface, specifically the one that List implements. While a list generally has some kind of ordering to its elements, the elements in a collection do not necessarily have an order.

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

	`System.out.println(arrayList.contains(3));` `System.out.println(arrayList.contains(-1));` `System.out.println(arrayList.containsAll(` `Arrays.asList(5,7));` `System.out.println(arrayList.containsAll(` `Arrays.asList(1,2,3));` `}` `}`
Output	`true` `false` `false` `true`
Explanation	A list is created and given five elements. Using the **contains()** method, we can verify that the list contains 3 and does not -1. Checking if the list contains 5 and 7 returns false since the list contains 5, but does not contain 7. Checking if the list contains 1, 2, and 3 returns true since 1, 2, and 3 are all in the list.

Removing groups of elements

- The List interface also has several methods for removing multiple elements at once. It is possible to remove all elements in a list or only remove elements that match specific criteria.

Method Signature	Description
void clear()	Removes all elements from the list, making it empty.
boolean isEmpty()	Returns true if the list is empty (it contains no elements), false otherwise.

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

	`` Arrays.asList(1,2,3,4,5)); System.out.println(arrayList); System.out.println(arrayList.isEmpty()); arrayList.clear(); System.out.println(arrayList); System.out.println(arrayList.isEmpty()); } } ```
Output	``` [1, 2, 3, 4, 5] false [] true ```

Method Signature	Description
boolean removeAll(Collection<?> c)	Removes all elements from the list that are contained in the provided collection. Returns true if any elements were removed, false otherwise.

Example	``` import java.util.*; public class Main { public static void main(String[] args) { List<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(1,2,3,4,4,5)); List<Integer> change = new ArrayList<Integer>(Arrays.asList(3,4)); arrayList.removeAll(change); System.out.println(arrayList); } } ```
Output	``` [1, 2, 5] ```

Explanation	A list is created and given six elements. Then a second list is created and **removeAll()** is used to remove all elements from the first list that were also in the second list. The original list had three elements that matched elements in the second list: 3, 4, and 4. With these elements removed, the list becomes [1, 2, 5]. Note that, unlike **remove(Object o)**, which only removes the first matching instance, **removeAll()** removes all instances that match any element in the provided collection. This is why, in the above example, both 4s are removed.

Method Signature	Description
boolean retainAll(Collection<?> c)	Removes all elements from the list that are *not* contained in the provided collection. Returns true if any elements were removed, false otherwise.

Example

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

    List<Integer> change = new ArrayList<Integer>(
        Arrays.asList(3,4));

    arrayList.retainAll(change);
    System.out.println(arrayList);
  }
}
```

Output

```
[3, 4, 4]
```

Explanation

A list is created and given six elements. Then a second list is created and **retainAll()** is used to remove all elements from the first list that were NOT in the second list. The original list had three elements that did not match elements in the second list: 1, 2, and 5. With these elements removed, the list becomes [3, 4, 4].

Converting a List into an array

- There may be times when it is useful to convert List objects into arrays. For instance, you may want to use the data in a List with a function that expects an array. Lists have a method called **toArray()** that returns the data in the list represented as a basic array.

Method Signature	Description
Object[] toArray()	Returns the contents of the List as an array.

Example

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

    Integer[] array = arrayList.toArray();
  }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
     Type mismatch: cannot convert from Object[] to
       Integer[]

     at edu.digipen.apcs.Main.main(Main.java:9)
```

Explanation

Because the **toArray()** method returns an array that contains Objects, not a specific type, we can't just assign it to an array of Integers. We will need to find a way to convert the array of Objects into an array of Integers.

Our first solution might be to try to directly cast the Object array to an Integer array like this:

```
Integer[] array = (Integer[]) arrayList.toArray();
```

However, this will only result in a ClassCastException, as Java cannot cast from an array of one type to an array of another type. We need to convert each element individually and store it in the new array.

Solution

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

    Object[] objArray = arrayList.toArray();
    Integer[] intArray = new Integer[objArray.length];

    for(int i = 0; i < objArray.length; ++i)
    {
      intArray[i] = (Integer)objArray[i];
    }
  }
}
```

Explanation

While the above solution works – it lets us convert the List of Integers to an array of Integers – it isn't as elegant as it could be. We need an extra variable for the array of objects, which we likely won't be using after we've filled the Integer array. Is there a way to shorten this?

As it turns out, Java provides another version of the toArray function that handles the type conversions for us.

Method Signature	Description
<T> T[] toArray(T[] array)	Fills the given array with the elements of the list converted to type T from type E. The array must be the same size as the List.

Example

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

    Integer[] intArray = new Integer[arrayList.size()];
    arrayList.toArray(intArray);
```

	`` for(Integer number : intArray) { System.out.print(number + " "); } System.out.println(); } } ```
Output	``` 1 2 3 4 5 ```

Iterators

- An iterator is an object that is used to traverse a list or other collection without knowing the particulars of how the collection is implemented.
- An iterator points to a specific element in the collection and can move forwards through the list (and sometimes backwards) from element to element.
- Basic iterators in Java have only three methods:

Method Signature	Description
boolean hasNext()	Returns true if there are additional elements in the collection after the element last returned by the iterator, false otherwise.
E next()	Returns the next element in the collection.
void remove()	Removes the last element returned by the iterator.

Using Iterators with Lists

- In order to use an iterator with a list, we first need to retrieve an iterator for the list from the list object. The List interface has several methods for getting iterators, the first of which is shown here:

Method Signature	Description
Iterator<E> iterator()	Returns an iterator starting just before the first element in the list. (The first call to next() will return the first element in the list.)

Example 1

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

    Iterator<Integer> it = arrayList.iterator();
    while(it.hasNext())
    {
```

	`` System.out.print(it.next() + " "); } System.out.println(); } } ```
Output	``` 1 2 3 4 5 ```
Explanation	After creating a list, an iterator is created by using the **iterator()** method. Notice that when the Iterator object is created, its type must be specified and must match the type of the List object. While the iterator can still move forward (the iterator's **hasNext()** method returns true), the next element in the list is retrieved using **next()** and is printed.

Example 2	``` import java.util.*; public class Main { public static void main(String[] args) { List<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5)); Iterator<Integer> it = arrayList.iterator(); while(true) { System.out.print(it.next() + " "); } } } ```
Output	``` 1 2 3 4 5 Exception in thread "main" java.util.NoSuchElementException at java.util.ArrayList$Itr.next(Unknown Source) at edu.digipen.apcs.Main.main(Main.java:12) ```
Explanation	If we attempt to retrieve the next element without checking to see if it exists, eventually we end up outside the bounds of the list which, since we're using an iterator to traverse the list for us, results in a *NoSuchElementException* on the next call to **next().**

Removing elements using Iterators

- Although the List interface does provide element removal functions, sometimes we may want to remove an element using more complex criteria than a simple equality check.
- For example, if we have a list of all the objects in our game and only want to perform logic for those that are still alive, we could use an iterator to traverse the list and remove only those objects that are considered "dead."

Example

```
/* GameObject Class */
public class GameObject
{
  private boolean dead;

  public GameObject()
  {
    dead = false;
  }

  public boolean isDead()
  {
    return dead;
  }

  public void kill()
  {
    dead = true;
  }
}
```

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

    System.out.println(arrayList.size());
    arrayList.get(0).kill();
```

```
    Iterator<GameObject> it = arrayList.iterator();
    while(it.hasNext())
    {
      GameObject go = it.next();
      if(go.IsDead())
      {
        it.remove();
      }
    }
    System.out.println(arrayList.size());
  }
}
```

Output

```
2
1
```

Explanation

The GameObject class has a field called **dead** which tells us whether the object is dead (true) or alive (false). Its initial value is true and can be changed to false by calling the **kill()** method. The **isDead()** method is used to access the current value of the **dead** field.

In the Main class in the main method, a list of GameObject instances is created. It is instantiated with two GameObjects, so its initial size is found to be 2.

The first element in the list is killed, which sets its **dead** field to true. An iterator is then created and used to traverse the list. While traversing the list, any element that is dead is removed using the iterator's **remove()** method.

After the traversal is complete, we find that the size of the list has been reduced to 1 since one of the GameObjects had been killed and was subsequently removed from the list.

Note: It is not possible to use **remove()** without first using **next()**. Attempting to call **remove()** on an iterator when the current element has already been removed will result in a *IllegalStateException*.

The ListIterator interface

- The ListIterator interface is similar to the Iterator interface, but provides additional methods for walking backwards through a list, as well as a few other operations.
- ListIterators can only be used with classes that implement the List interface. The List interface provides two functions for creating ListIterators:

Method Signature	Description
ListIterator<E> listIterator()	Returns a ListIterator starting just before the first element in the list.
ListIterator<E> listIterator(int index)	Returns a ListIterator starting at the given index.

- The methods used for backwards traversal through a list are **hasPrevious()** and **previous()** :

Method Signature	Description
boolean hasPrevious()	Returns true if there are additional elements in the list *before* the element last returned by the iterator, false otherwise.
E previous()	Returns the next element in the list.

Example

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

    ListIterator<Integer> it =
        arrayList.listIterator(arrayList.size());

    while(it.hasPrevious())
    {
      System.out.print(it.previous() + " ");
    }
    System.out.println();
  }
}
```

Output	5 4 3 2 1
Explanation	In this example, a ListIterator is used to walk backwards through a list, printing each element. After creating a list, an iterator is created by using the **listIterator(int index)** method. The index given as the iterator's starting point is **arrayList.size()**. Is it okay to use this as a starting index? Like arrays, lists begin at 0, so **size()** is actually one past the last element in the list. Normally, starting here would be a problem. However, since we're going backwards through the list using **previous()**, if we want to hit each element, we need to start at one past the end. This way, the first call to **previous()** will give us the last element in the list. Before moving the iterator to the previous element, **hasPrevious()** is called to make sure a previous element actually exists. Once there are no more elements to iterate through (**hasPrevious()** returns false), the loop stops.

Adding and replacing elements with ListIterator

- The ListIterator interface also has methods for adding and replacing elements while traversing through a list:

Method Signature	Description
void add(E e)	Inserts the specified element into the list. This element will be inserted immediately before the next element in the list.

Example

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

    ListIterator<Integer> it =
        arrayList.listIterator(arrayList.size());

    it.previous();
```

	`it.add(42);` `System.out.println(arrayList);` `}` `}`
Output	`[1,2,3,4,42,5]`
Explanation	In this example, a ListIterator is used to add an element to the list. How do we know where the element will be inserted? It will be inserted before the next element, the element that would be returned by the next call to **next()**. After one call to **previous(),** the next call to **next()** would give us the last element in the list, 5. Thus, 42 is inserted just before the last element, between 4 and 5.

Method Signature	Description
void set(E e)	Replaces the element returned by the last call to next() or previous() (whichever was called most recently).

Example 1

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

    ListIterator<Integer> it =
        arrayList.listIterator(arrayList.size());

    it.previous();
    it.set(42);
    System.out.println(arrayList);
  }
}
```

Output

```
[1,2,3,4,42]
```

Explanation	In this example, a ListIterator is used to replace an element to the list. How do we know which element will be replaced? To figure this out, we need to know where the iterator has been and, most importantly, which element would have been returned by the most recent call to **next()** or **previous()**. The most recent iteration was done using **previous()**. Since the iterator began at one past the last element of the list and hadn't moved prior to the most recent iteration, that call to **previous()** would have returned the last element in the list, which was 5. This element is then replaced by the value given to the **set()** method, 42.

Example 2	`import java.util.*;` `public class Main` `{` `  public static void main(String[] args)` `  {` `  List<Integer> arrayList = new ArrayList<Integer>(` `  Arrays.asList(1,2,3,4,5));` `  ListIterator<Integer> it =` `  arrayList.listIterator();` `  it.next();` `  it.set(42);` `  System.out.println(arrayList);` `  }` `}`
Output	`[42,2,3,4,5]`
Explanation	This time, the most recent iteration before the call to **set()** was done using **next()**. Since the iterator's position before the call to **next()** was just before the first element in the list, that call to **next()** would have returned the first element, which was 1. This element is then replaced by the value 42.

- Attempting to replace an element before **next()** or **previous()** have been called will result in an *IllegalStateException*.
- An exception will also occur if **set()** is called directly following a **remove()** operation.

Example 3	`import java.util.*;` `public class Main` `{` `  public static void main(String[] args)` `  {` `  List<Integer> arrayList = new ArrayList<Integer>(` `  Arrays.asList(1,2,3,4,5));` `  ListIterator<Integer> it =` `  arrayList.listIterator();` `  it.next();` `  it.remove();` `  it.set(42);` `  System.out.println(arrayList);` `  }` `}`
Output	`Exception in thread "main" java.lang.IllegalStateException` `  at java.util.ArrayList$ListItr.set(Unknown Source)` `  at edu.digipen.apcs.Main.main(Main.java:13)`
Explanation	Although **next()** was called before using **set()**, the element that would have been returned by **next()** was removed and therefore no longer exists. It is impossible to replace something that does not exist, so an exception is thrown.