CSY2030
Systems Design & Development

Collections, ArrayLists and Generics

Overview of Lecture

- Today we will look at the following in Java:
  - Collections
  - Motivation for ArrayLists
  - ArrayLists
  - Generics

Collections

- A Collection is a group of objects, called *elements*
  - Collections are defined as classes
- There are different types of collections e.g
  - **List -** an ordered collection, duplicates are allowed
    - **ArrayList** is an implementation of List (which is an interface)
  - **Set -** an unordered collection with no duplicates
    - **SortedSet -** an ordered collection with no duplicates
  - **Map -** a collection that maps *keys* to *values*
    - **SortedMap -**a collection ordered by the keys

Motivation for ArrayLists

- We need to store many items
  - This can be done with arrays
- However, arrays have many shortcomings:
  - Array sizes cannot be changed
  - When deleting an item you have spaces in the array
    - Reorganising is cumbersome
  - Many things you want to do with arrays e.g adding/deleting/modifying have to be re-done for each type of array
    - Can't change type of an array
- To overcome these problems Java provides **ArrayLists**

ArrayLists

- An ArrayList is a collection that stores data like an array
- Like an array, every element is stored in an integer index that stores its position in the sequence
- Every element in the list has a numerical index
  - The first element in the list has index = 0
- However, unlike an array it can store any number of elements and the number of elements do not have to be declared when the list is created

ArrayLists

- Methods of ArrayList include the following:
  - **add(Object obj)**
    - Appends the object obj to the end of this ArrayList
  - **add(int index, Object element)**
    - Inserts the element at position index in this ArrayList
  - **remove(Object obj)**
    - Removes the first occurrence of obj from this ArrayList
  - **remove(int index)**
    - Removes the element at position index from this ArrayList
  - **clear()**
    - Removes all elements
  - **get(int index)**
    - Returns the component at position index
  - **set(int index, Object element)**
    - replaces the element at the specified position in this list with the specified element.
  - **contains(Object element)**
    - Tests if element is a component of this ArrayList
  - **indexOf(Object element)**
    - Returns the index of the first occurrence of element in this ArrayList
  - **isEmpty()**
    - Returns true if this ArrayList has no elements
  - **size()**
    - Returns the number of elements currently in this ArrayList

ArrayLists Example 1

```
import java.util.ArrayList;

public class ArrayListDemo1
{
        public static void main(String[] args)
        {
            ArrayList nameList = new ArrayList();

            nameList.add("James");
            nameList.add("Catherine");
            nameList.add("Bill");

            System.out.println("The ArrayList has " +  nameList.size()
                                                + " objects stored in it.");

            for (int index = 0; index < nameList.size(); index++)
                        System.out.println(nameList.get(index));
        }
}
```

Needed for ArrayList class

Create an ArrayList to hold some names.

Add some names to the ArrayList.

Display the size of the ArrayList.

Now display the items in nameList.

Output

```
The ArrayList has 3 objects stored in it.
James
Catherine
Bill
```

ArrayLists Example 2

```
import java.util.ArrayList;
public class ArrayListDemo2
{
	public static void main(String[] args)
	{
		ArrayList nameList = new ArrayList();
		nameList.add("James");
		nameList.add("Catherine");
		nameList.add("Bill");

		for (int index = 0; index < nameList.size(); index++) {
			System.out.println("Index: " + index + " Name: " + nameList.get(index));
		}

		nameList.remove(1);

		System.out.println("The item at index 1 is removed. " +  "Here are the items now.");
		for (int index = 0; index < nameList.size(); index++) {
			System.out.println("Index: " + index + " Name: " + nameList.get(index));
		}
	}
}
```

Remove element at index 1

Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
The item at index 1 is removed. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Bill
```

ArrayLists Example 3

```
import java.util.ArrayList;
public class ArrayListDemo3
{
        public static void main(String[] args)
        {
                ArrayList nameList = new ArrayList();
                nameList.add("James");
                nameList.add("Catherine");
                nameList.add("Bill");

                for (int index = 0; index < nameList.size(); index++) {
                  System.out.println("Index: " + index + " Name: " + nameList.get(index));
                }

                nameList.add(1, "Mary");

                System.out.println("Mary was added at index 1. " + "Here are the items now.");

                for (int index = 0; index < nameList.size(); index++)   {
                        System.out.println("Index: " + index + " Name: " + nameList.get(index));
                }
        }
}
```

Now insert item at index 1

Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
Mary was added at index 1. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Mary
Index: 2 Name: Catherine
Index: 3 Name: Bill
```

ArrayLists Example 4

```
import java.util.ArrayList;
public class ArrayListDemo4
{
        public static void main(String[] args)
        {
                ArrayList nameList = new ArrayList();
                 nameList.add("James");
                nameList.add("Catherine");
                nameList.add("Bill");

                for (int index = 0; index < nameList.size(); index++) {
                        System.out.println("Index: " + index + " Name: " + nameList.get(index));
                }

                nameList.set(1, "Becky");

                System.out.println("Catherine was replaced with Becky. " +  "Here are the items now.");
                for (int index = 0; index < nameList.size(); index++) {
                        System.out.println("Index: " + index + " Name: " + nameList.get(index));
                }
         }
}
```

Now replace item at index 1

Output

Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
Catherine was replaced with Becky. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Becky
Index: 2 Name: Bill

ArrayList Base Type

- You can specify the type of each element in the ArrayList
- The base type of an ArrayList is specified as a *type parameter*
- Base type is declared as follows:

```
ArrayList<BaseType> aList =
                    new ArrayList<BaseType>();
```

ArrayLists Example 5

```
import java.util.ArrayList;
public class ArrayListDemo5
{
        public static void main(String[] args)
        {
            ArrayList<String> nameList = new ArrayList<String>();

            nameList.add("James");
            nameList.add("Catherine");
            nameList.add("Bill");

            System.out.println("The ArrayList has " + nameList.size() + " objects stored in it.");

            for (int index = 0; index < nameList.size(); index++)
                    System.out.println(nameList.get(index));
        }
}
```

Specify base type as String

Output

The ArrayList has 3 objects stored in it.
James
Catherine
Bill

Generics

- Java provides a mechanism for allowing classes to work with any type
- This is called a *Generic*
- Generics can be used to overcome the problem of repeated code and allow methods to be used on any type

Defining Generics

- When declaring a class as a Generic you use Angle Brackets.
- The class is defined using the header

```
public class List<T> {
```

- The <T> in brackets can be thought of as a variable name. You can call this anything you like

Defining Generics

- You can then use the variable name in your class anywhere you would normally use a type:

```
public class List<T> {
        private Object[] items = new Object[2];
        private int length = 0;

        public void add(T item) {
                if (length == items.length)
                        resize();
                this.items[this.length] = item;
                this.length++;
        }


        public T get(int index) {
                return (T)this.items[index];
        }
}
```

Generics

- This allows you to define the type when you initialise the list inside the angle brackets:

```
List list = new List<String>();

list.add("A");
list.add("B");
list.add("C");
list.add("D");

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

```
List list = new List<Integer>();

list.add(1);
list.add(2);
list.add(3);
list.add(4);

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

```
List list = new List<Boolean>();

list.add(false);
list.add(true);
list.add(true);
list.add(false);


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

```
List list = new List<Double>();

list.add(12.3);
list.add(9.887);
list.add(22.3);
list.add(1.093);

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

Generics

- This allows you to define the class once and reuse it with any type
- This has the advantage of making your class more reusable
- However there is one caveat: Generics only support objects! Note that the type in the angle brackets start with uppercase letters!

```
List list = new List<String>();
List list = new List<Integer>();
List list = new List<Boolean>();
List list = new List<Double>();
```

Generics Example

- Consider a box class that represents a rectangle with a width and a height
- You may not know the dimensions being used up front

Generics Box Class

```
public  class  Box<T>  {
    private   T   height;
    private T width;

    public Box(T width, T height) {
        this.width = width;
        this.height = height;
    }
    public T getWidth() {
                return width;
    }
    public T getHeight() {
                return height;
    }
    public String toString() {
             return "Width: " + width + " height: " + height;
    }
}
```

The constructor uses the Generic type

Generic box class

- This way, the box class can be used with any type:

```
Box<Integer> box1 = new Box<Integer>(4, 5);

System.out.println(box1.toString());
```

Because the constructor is defined as requiring type T, You must pass arguments that match the Generic type (in this case Integer)

```
Output:
Width: 4 height: 5
```

Generic box class

- This way, the box class can be used with any type:

```
Box<Double> box1 = new Box<Double>(4.2, 5.3);

System.out.println(box1.toString());
```

By changing the generic Type, the arguments Must also be changed

```
Output:
Width: 4.2 height: 5.3
```

Generic box class

- This way, the box class can be used with any type:

```
Box<String> box1 = new Box<String>("14cm", "22cm");

System.out.println(box1.toString());
```

You could do the same with Strings

```
Output:
Width: 14cm height: 22cm
```

Generic names

- The type T is used by convention but you can use anything.
- The conventions are as follows:

Name	Usual Meaning
T	Used for a generic type.
S	Used for a generic type.
E	Used to represent generic type of an element in a collection.
K	Used to represent generic type of a key for a collection that maintains key/value pairs.
V	Used to represent generic type of a value for collection that maintains key/value pairs.

Generic objects as arguments

- When declaring a method argument that takes a generic type, you **must** provide the full class name including the type

```
public static void main(String[] args) {
    Box<String> box1 = new Box<String>("12cm", "4cm");
    showBoxWidth(box1);
}

public static void showBoxWidth(Box<String> box) {
    System.out.println("Box has a width of " +   box.getWidth());
}
```

Generic objects as arguments

- However, arguments only work with the provided types, this will cause an error

```
public static void main(String[] args) {
     Box<String> box1 = new Box<String>("12cm", "4cm");
     showBoxWidth(box1);
}

public static void showBoxWidth(Box<Integer> box) {
     System.out.println("Box has a width of " +    box.getWidth());
}
```

Wildcard parameters

- There is a wildcard parameter which allows you to pass in a generic of any type.
- Using a **?** character inside the angle brackets allows parameters of any generic type to be used

```
public static void main(String[] args) {
    Box<String> box1 = new Box<String>("12cm", "4cm");
    showBoxWidth(box1);
    Box<Integer> box2 = new Box<Integer>(12, 4);
    showBoxWidth(box2);
}

public static void showBoxWidth(Box<?> box) {
    System.out.println("Box has a width of " + box.getWidth());
}
```