**Abstract Data Types**

- A set of values and operations that can be applied to underlying data.
- Can describe the concepts or capabilities in terms of their functionality, not their implementation details.
- Classes can be considered Abstract Data Types of objects you want to represent.
  - Public methods for these classes provide functionality to manipulate the underlying data.
  - Think of the String class….
    - They're a higher-level concept of a sequence of characters.
    - The implementation details of strings are hidden from the users, such as memory allocation, maintaining the length, …

**Lists – Abstract Data Types**

- We've talked about the general need to store objects in a collection.
- ArrayLists use an array as the underlying mechanism to maintain this information.
  - We cannot modify the actual array directly.
  - We will need to use public methods such as .size(), .contains(item),. clear(), .add(i, item), .remove(i), etc.
  - And we probably shouldn't be able to manipulate the data directly…
    - It adds complexity for the consumer of the ArrayList class.
    - Programmers may accidentally break the implementation if they were allowed direct access.
- LinkedList is a class similar to ArrayList
  - It even has most of the methods that an ArrayList has.
  - But the "under-the-hood" implementation is completely different.
    - Arrays are not used, but a chain of objects referring to each other is how LinkedLists are implemented.
- Both LinkedLists and ArrayLists are Abstract Data Types
  - Both hide the implementation details.
  - For simple tasks, not knowing the underlying details to complete a task is good enough…
    - Why would you care about knowing the underlying details of a Scanner, PrintWriter, System.out, … ?
  - Though, it's important for software developers to know what goes on under-the-hood.
    - Programming is an art and different problems require different solutions.
    - It prevents a programmer from producing inefficient code depending on the problem.
      - For example, some implementations may benefit using a LinkedList more than using an ArrayList (and vice-versa).

**LinkedList Example**

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

list.add(5);
list.add(10);
list.add(15);
list.add(20);

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

- Replace LinkedList with ArrayList and see they are interchangeable.

**Common LinkedList Methods**

- `list.remove(1); // removes 10 from the list`
- `list.remove(new Integer(10));//removes first occurrence of 10.`
- `list.add(25); // adds to end of list (also list.addLast(item))`
- `list.addFirst(0); // adds to head of the list`
- `System.out.println(list.contains(25)); // checks if item exists`
- `list.clear(); // removes all elements in list.`

**Performance Analysis of ArrayList vs. LinkedList**

- ArrayList (AL) vs. Singly-linked list (LL) with a head reference
    - Insertion at the beginning of AL: O(n), LL: O(1)
    - Insertion at the end of AL: O(1), LL: O(n)
    - Removing at the beginning of AL: O(n), LL: O(1)
    - Removing at the end of AL: O(1), LL: O(n)
    - Random access of AL: O(1), LL: O(n)
    - Random search of AL: O(n), LL: O(n)

- Java's implementation is actually a doubly-linked list with a head AND tail reference.
    - Insertion at the beginning: O(1)
    - Insertion at the end: O(1)
    - Removing at the beginning: O(1)
    - Removing at the end: O(1)
    - Random access: O(n)
    - Random search: O(n)

**Syntactic Sugar**

- We can simplify our for loops with a special "for each" syntax

```
for (int x : list) {
      System.out.println(x);
}
```

- Only works for objects that implement the Iterable Interface.
    - Which includes ArrayLists and LinkedLists!

- Be careful!
    - If modifications are made to the list while traversing it, an ERROR **may** occur.

**Generics**

- We've seen how Generics can be used with LinkedLists and ArrayLists.
- Let's talk about how a programmer can create their own generic representations.

**Generic Methods**

- Assume we want to get the last item of an array and return that item.
- We could write the same method for all different objects?
    - Crazy....
- We can write a generic method to account for ALL types of possible arrays and return that specific type.

**Example**

```
public static <T> T getLastItem(T[] array) {
      if (array.length > 0) {
            return array[array.length - 1];
      }
      return null;
}

Integer[] intArray = {1,2,3}; // int[] error, requires Integer
Double[] doubleArray = {1.1, 2.2, 3.3};
String[] stringArray = {"I", "<3", "ICS45J"};

System.out.println(getLastItem(intArray));
System.out.println(getLastItem(doubleArray));
System.out.println(getLastItem(stringArray));
```

**Generic Classes**

- We can also create entire classes that are generic.
- Assume we want to write a simple class storing a pair of values (both of the same type).

```
public class Pair<T> {
      private T first;
      private T second;

      public Pair(T first, T second) {
            this.first = first;
            this.second = second;
      }

      public T getFirst() {
            return first;
      }

      public T getSecond() {
            return second;
```

```
	}

	public void print() {
		System.out.println(first + ", " + second);
	}
}

Pair<Integer> pair = new Pair<Integer>(1,2);
System.out.println(pair.getFirst() + pair.getSecond());
pair.print();
```

- The above class assumes first and second values are of the same type.
- We may not want to make that restriction and attempt to store a Pair where first and second can be of any Object type.
- When the Pair object is constructed, the appropriate type for first and second.

```
public class Pair<T,U> {
	private T first;
	private U second;

	public Pair(T first, U second) {
		this.first = first;
		this.second = second;
	}

	public T getFirst() {
		return first;
	}

	public U getSecond() {
		return second;
	}

	public void print() {
		System.out.println(first + ", " + second);
	}
}

Pair<Integer, String> p1 = new Pair<Integer,String>(0,"Richert");
Pair<Integer, String> p2 = new Pair<Integer,String>(1, "Mr. E");

System.out.println(p1.getFirst() + " - " + p1.getSecond());
System.out.println(p2.getFirst() + " - " + p2.getSecond());

p1.print();
p2.print();
```

**2D Arrays**

- Arrays we've been talking about so far have been a one dimensional (i.e. a simple list).
- We can actually organize data into multiple dimensions with multi-dimensional arrays.
- A good way to think of it is an array of arrays…

**Example**

- Create a 4x5 grid of int values.

```
int[][] int2d = new int[4][5];

// traverse the entire 2D array structure and print it out in a matrix
for (int i = 0; i < int2d.length; i++) {
      for (int j = 0; j < int2d[i].length; j++) {
            System.out.print(int2d[i][j] + " ");
      }
      System.out.println();
}
```

- We've seen something similar to this when creating an ArrayList containing ArrayLists.
- We can set / get values using [x][y] notation.
- x represents the row while y represents column.
- Remember the game battleship??

```
int2d[1][3] = 8;
```

0 0 0 0 0
0 0 0 8 0
0 0 0 0 0
0 0 0 0 0

Initialize values during construction:

```
int [][] int2d =
{{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20}};
```