Throws

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Syntax of Java throws

```
1.return_type method_name() throws exception_class_name{

1.    //Method Code
2.}
```

```
  public void someMethod(int value) throws IllegalArgumentException{

  }
```

Throw

Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.

Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only

The throw keyword is followed by an instance of Exception to be thrown.

Syntax of Java throw

```
if (num < 1) {
  throw new {Exception Class}()
}
```

```
public void someMethod(int value) {
    if (value < 0) {
     throw new IllegalArgumentException();
    }
   // Rest of the method's logic
}
```

Custom Exception Class

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Example -01

- Using Java Inheritance implement the flowing calculator generations

- The Calculator class defines basic arithmetic operations add, subtract, multiply, divide
- The DegitalCalculator has additional functions like square root and power.
- The GraphingCalculator New functionality of Plotting a graph other than all functionalities of the

- Note : Does not require actual plot implementation use a print statement is sufficient.
- Challenge 01 -> add function should be able to add two numbers or three numbers
- Challenge 02 -> In the GraphCalulator new multifaction method implement only using add operation only ?

Example -02

Let's consider an interface Shape that defines the common methods for various geometric shapes.

```
interface Shape {
 double calculateArea();
 double calculatePerimeter();
}
```

Write a Program for Circle and Rectangle classes implement this interface, providing their own implementations for the methods

Circle – Area = PI * radius * radius

Circle – Perimeter = 2 * PI * radius

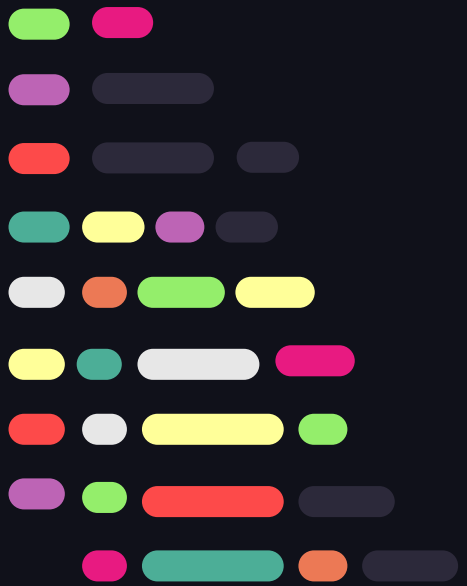
Rectangle – Area = width * height

Rectangle – Perimeter = 2 *(width + height)

Java Programming Week-05

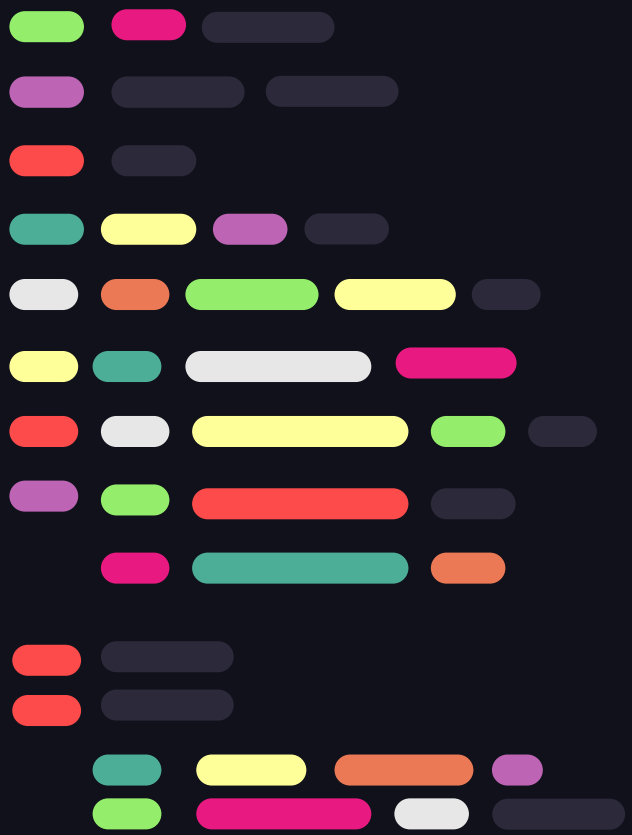
<Mohan De Zoysa –IIT JAVA CERTIFICATION>

Outline



Java Collection Framework

- Introduction
- Working with arrays and arrayLists
- Working with LinkedLists
- Working with HashSet
- Working with HashMap
- Using iterators and enumerations



Collection in Java

A Collection represents a single unit of objects, i.e., a group.

framework in Java

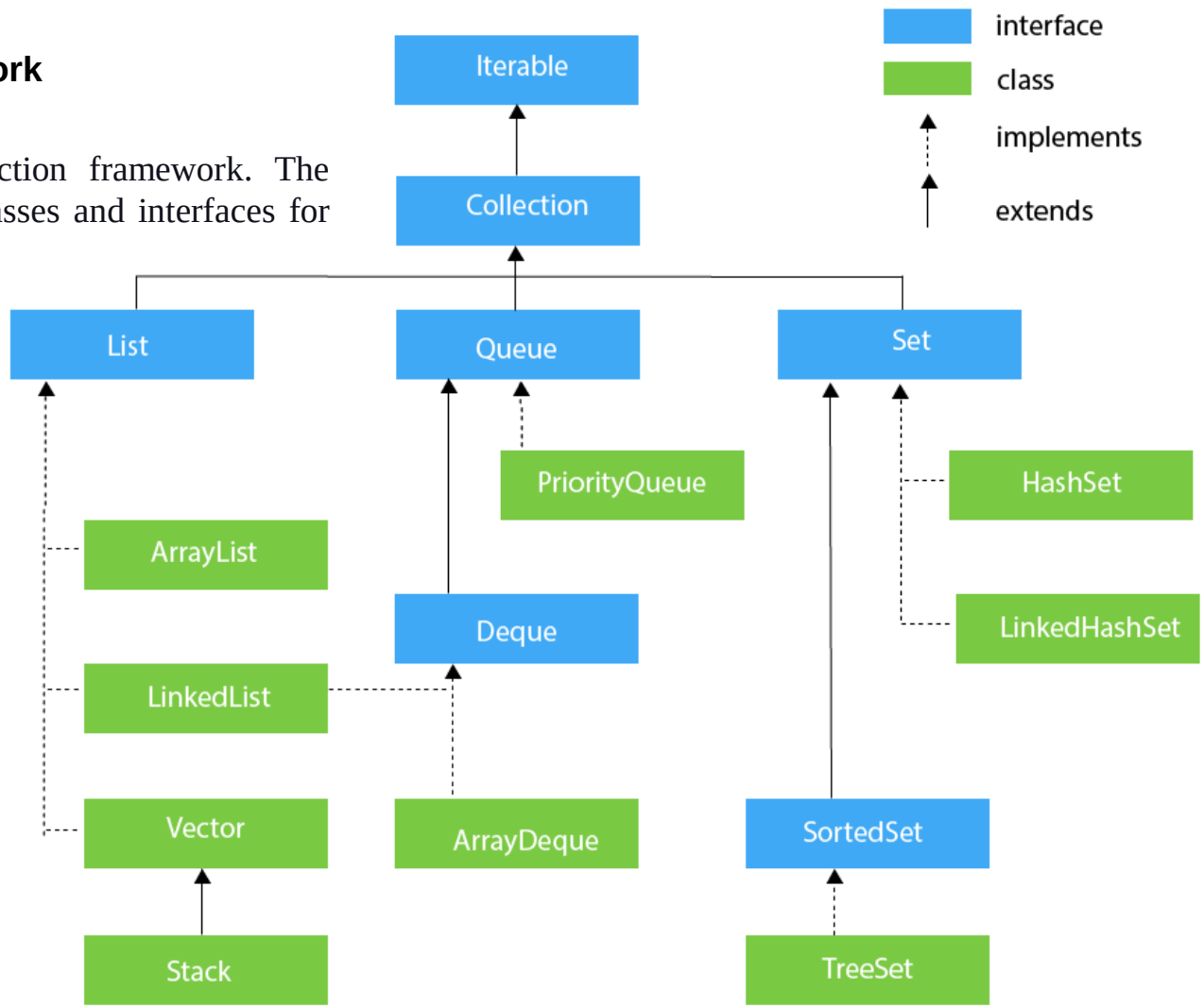
In Java, a framework refers to a reusable and structured set of libraries, classes, and tools that provide a foundation for developing various types of software applications

Java Collection Framework

The Java Collections Framework is a comprehensive set of classes and interfaces that provides a standardized way to manage and manipulate groups of objects. It offers a wide range of data structures and algorithms for storing, organizing, and accessing data efficiently. The collections framework is part of the java.util package and is a fundamental aspect of Java programming.

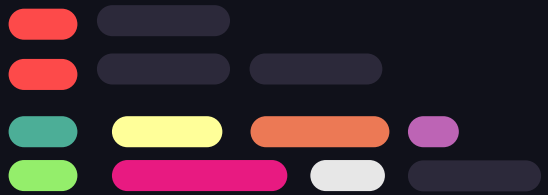
Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.

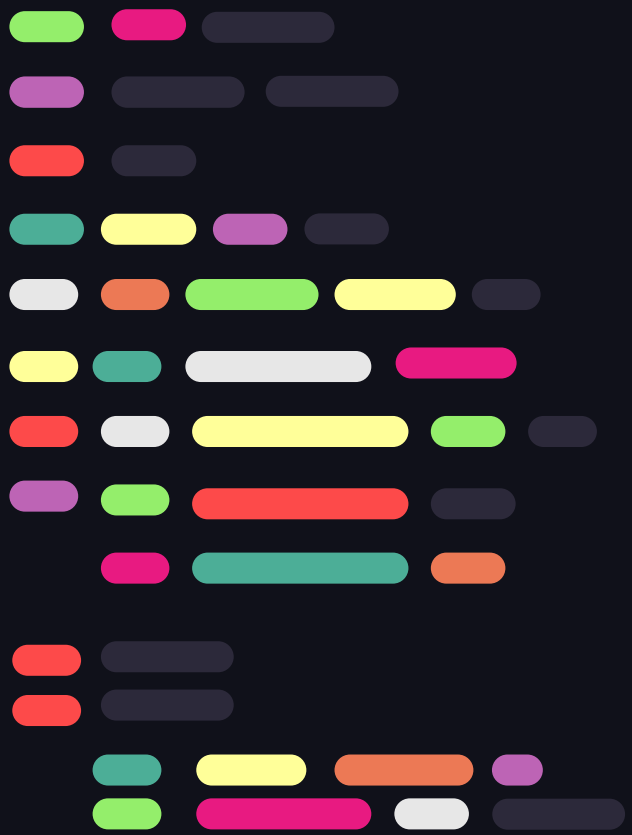


{..

ArrayLists



..}



Java ArrayList

In Java, an ArrayList is used to represent a dynamic list.

While Java arrays are fixed in size (the size cannot be modified), an ArrayList allows flexibility by being able to both add and remove elements.

Create an ArrayList

```
// import the ArrayList Class
import java.util.ArrayList;

// create an ArrayList called students
ArrayList<String> students = new ArrayList<String>();
```

Before Java 7, you would have needed to provide the type on both sides of the assignment, like this:

Starting from Java 7, the Diamond Operator can be used to simplify the syntax, making it more concise and readable.

```
ArrayList<String> students = new ArrayList<>();
```

Modifying ArrayLists in Java

An ArrayList can easily be modified using built in methods.

To add elements to an ArrayList, you use the add() method. The element that you want to add goes inside of the ().

To remove elements from an ArrayList, you use the remove() method.

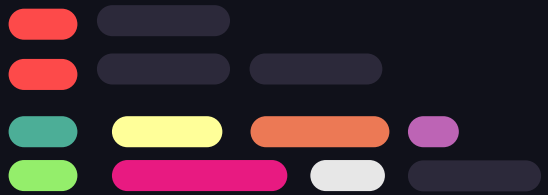
Inside the () you can specify the index of the element that you want to remove. Alternatively, you can specify directly the element that you want to remove.

Modifying ArrayLists in Java cont.

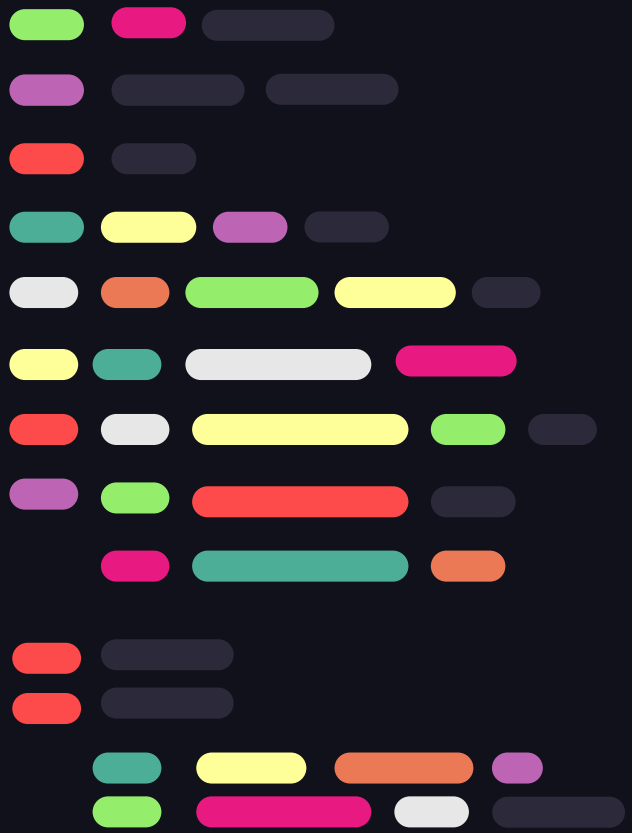
```
// create an ArrayList called studentList, which initially holds []
     ArrayList<String> studentList = new ArrayList<String>();

    // add students to the ArrayList
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the ArrayList, then Lily
    studentList.remove(0);
    studentList.remove("Lily");
```

{ . .

LinkedLists



} . .



Java LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Create a LinkedList

```
// import the LinkedList Class
import java.util.LinkedList;

// create an ArrayList called students
LinkedList <String> linkedList =new LinkedList<>();
```

Modifying LinkedLists in Java

An LinkedLists can easily be modified using built in methods.

To add elements to an LinkedLists, you use the add() method. The element that you want to add goes inside of the ().

To remove elements from an LinkedLists, you use the remove() method.

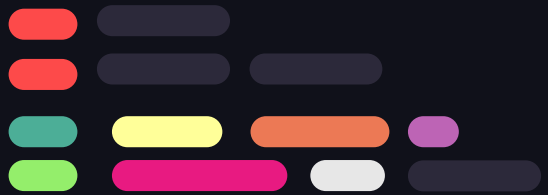
Inside the () you can specify the index of the element that you want to remove. Alternatively, you can specify directly the element that you want to remove.

Modifying LinkedList in Java cont.

```
// create an LinkedList called studentList, which initially holds []
     LinkedList <String> studentList =new LinkedList<>();

    // add students to the LinkedList
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the LinkedList, then Lily
    studentList.remove(0);
    studentList.remove("Lily");
```

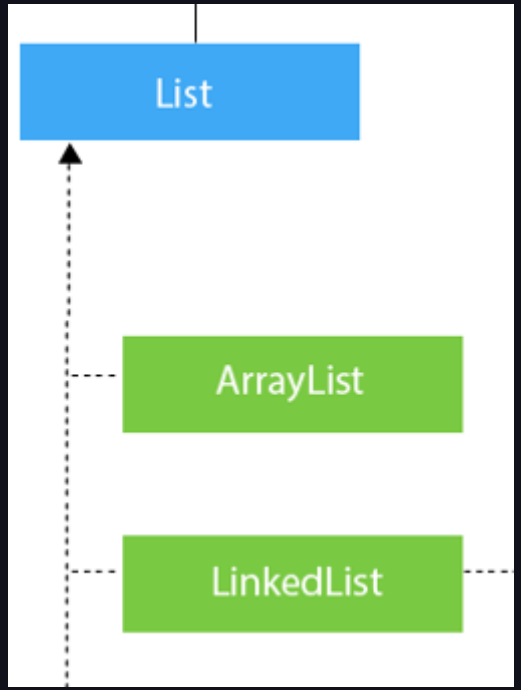
{ . .

ArrayList vs LinkedList



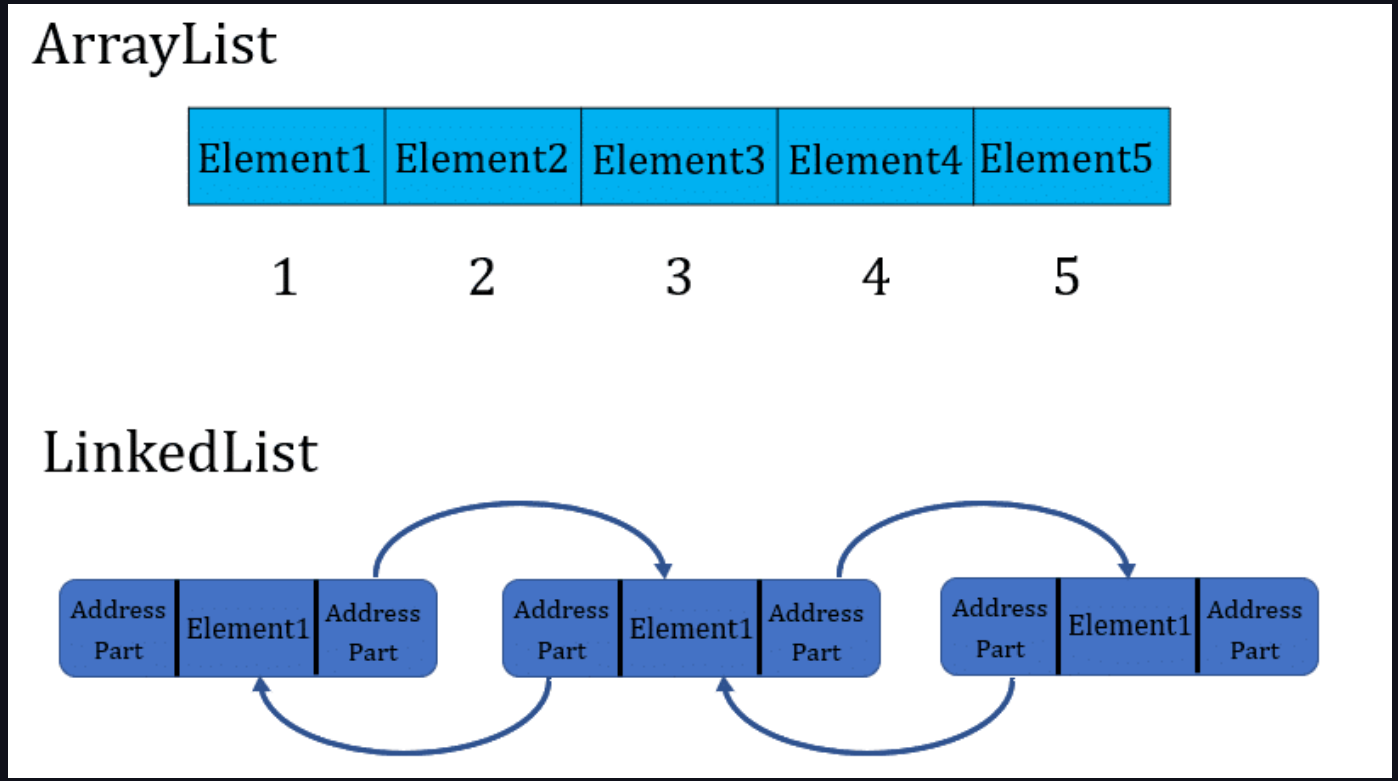
} . .

Array List & Linked List Hierarchy



Both LinkedList and ArrayList are implementations of the List interface in Java, but they have different characteristics due to their underlying data structures.

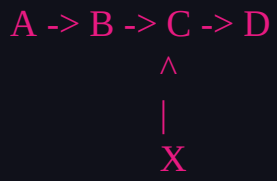
ArryList & LinkedList



Adding a new item in Linked List

Original Linked List: A -> B -> C -> D

1. Traverse to the target index (2).
2. Create a new node (X).



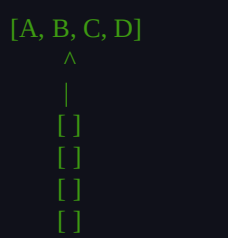
3. Update references:
   - Set X.next = C
   - Set B.next = X

Final Linked List: A -> B -> X -> C -> D

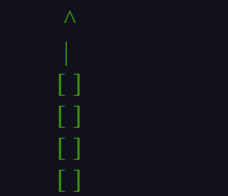
Adding a new item in Arry List

Original ArrayList: [A, B, C, D]

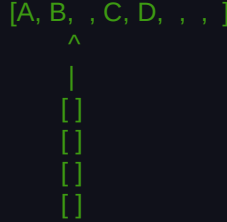
1. Adding at index 2 (item X).
2. Check if resizing is needed (current size is 4, capacity is 4, resizing needed).



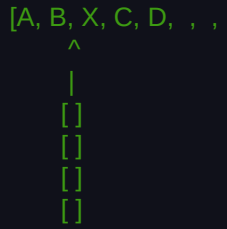
3. Resize the array (double the capacity, new size is 8).



4. Shift elements to make space for the new item.



5. Insert the new item (item X) at index 2.



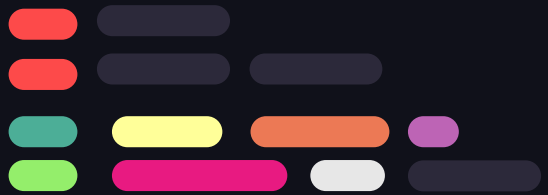
Final ArrayList: [A, B, X, C, D]

Comparison ArrayList & LinkedList

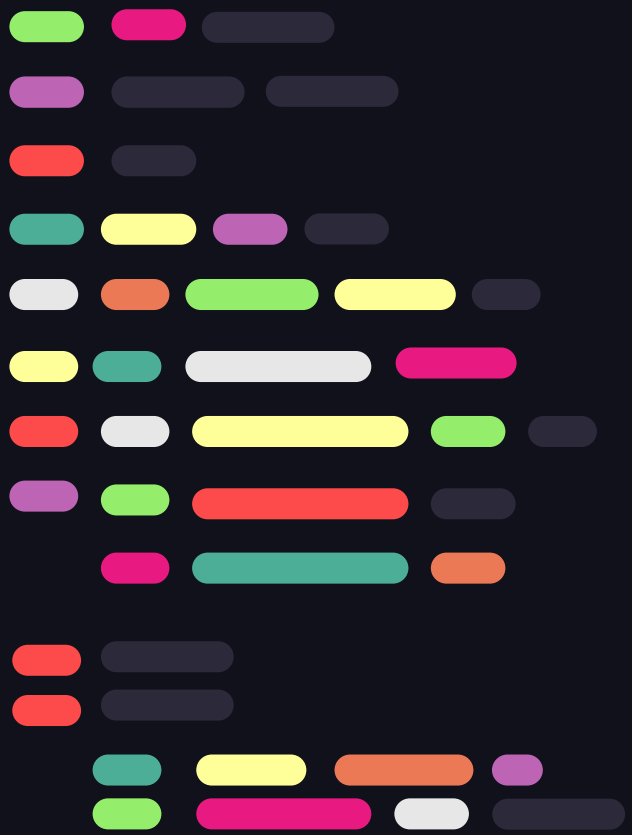
Factor	ArrayList	LinkedList
Underlying Data Structure	Dynamic array	Doubly linked list
Random Access (get by index)	O(1)	O(n)
Insertions/Deletions (middle)	Slower due to shifting elements	Faster due to adjusting pointers
Insertions/Deletions (beginning/end)	Slower due to shifting elements	Faster due to adjusting pointers
Memory Usage	Less memory per element	More memory per element
Resizing	Requires occasional resizing	No resizing needed
Iterating	Slightly faster due to contiguous memory access	Slightly slower due to linked traversal
Iterators	Fail-fast (ConcurrentModificationException)	Fail-fast (ConcurrentModificationException)
Search/Containment	O(n)	O(n)
Performance Trade-offs	Better for read-heavy operations	Better for write-heavy and mid-insertions
Data locality	Better cache locality due to contiguous memory	Poorer cache locality due to non-contiguous
Space Complexity	O(n)	O(n)
Use Cases	Read-heavy, random access	Write-heavy, frequent insertions/deletions
Example Use	Storing database records, static data	Implementing queues, circular lists, etc.

{..

HashSet



}..



HashSet

A HashSet in Java is a collection class that implements the Set interface. It represents an unordered collection of unique elements, where duplicate elements are not allowed. The HashSet is based on the hash table data structure, which allows for fast and efficient insertion, deletion, and lookup operations.

Create a HashSet

```
// import the LinkedList Class
import java.util.HashSet;
import java.util.Set;


// create an HashSet with interface type
Set<String> hashSet =new HashSet<>();

// create an HashSet with Using Concrete Type

 HashSet<String> hashSet =new HashSet<>();
```

Modifying HashSet in Java

An HashSet can easily be modified using built in methods.

To add elements to an HashSet, you use the add() method. The element that you want to add goes inside of the ().

To remove elements from an HashSet, you use the remove() method.

However, you cannot specify the index of the element that you want to remove.

Important HashSet not grantee the order of the item which you used to store

HashSet Not store duplicate values

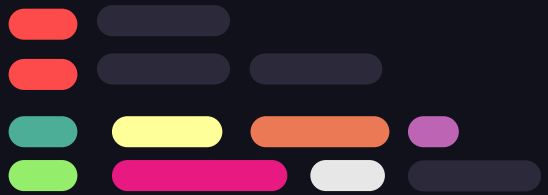
Super fast with adding and removal

Modifying LinkedList in Java cont.

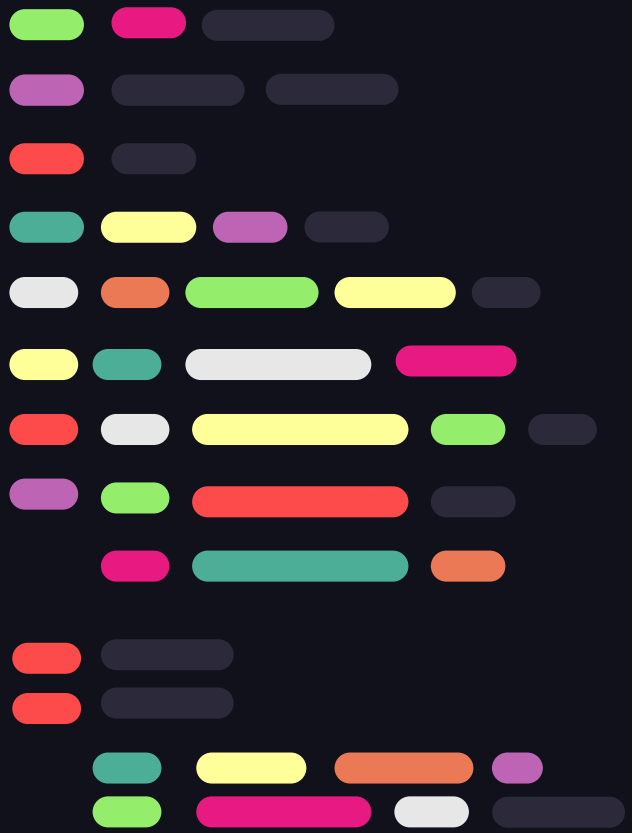
```
// create an LinkedList called studentList, which initially holds []
    Set<String> hashSet =new HashSet<>();

    // add students to the LinkedList
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the LinkedList, then Lily
    studentList.remove(0); // this remove method will not make any effects
    studentList.remove("Lily");
```

{..

HashMap



..}



HashMap

A HashMap in Java is a collection class that implements the Map interface. It stores elements as key-value pairs, where each key is associated with a value. HashMap is based on the hash table data structure, which allows for fast and efficient retrieval, insertion, and deletion of key-value pairs.

Create a HashMap

```
// import the LinkedList Class
import java.util.HashMap;


// create an HashMap
HashMap<String,Integer> employeesMap =new HashMap<>();
```

Modifying HashMap in Java

An HashSet can easily be modified using built in methods.

To add elements to an HashMap, you use the put() method. We use Key , Value pair

To remove elements from an HashMap, you use the remove() method.

However, you cannot specify the index of the element that you want to remove.

Important HashMap not grantee the order of the item which you used to store

HashSet Not store duplicate values

Super fast with adding and removal

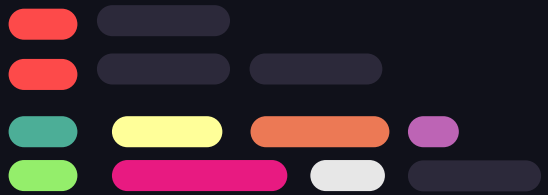
Can access form both key and value

Modifying HashMap in Java cont.

```
HashMap<String, Integer> scores = new HashMap<>();

scores.put("Alice", 90);
scores.put("Bob", 85);
scores.put("Charlie", 92);

System.out.println(scores.get("Alice")); // Output: 90

scores.remove("Bob");
System.out.println(scores); // Output: {Charlie=92, Alice=90}
```

{..

Using iterators



..}

Iterating

Two possible methods

```
//Using foreach
for (String name: linkedList) {
            System.out.println(name);

}

//Using Iterator

Iterator<String> iterator = linkedList.iterator();
while (iterator.hasNext() ){
   System.out.println(iterator.next());

}
```

Iterating cont.

```
// Get an iterator for the entry set of the map

Iterator<Map.Entry<String, Integer>> iterator = scores.entrySet().iterator();

// Iterate through the key-value pairs
while (iterator.hasNext()) {
            Map.Entry<String, Integer> entry = iterator.next();
            String name = entry.getKey();
            int score = entry.getValue();
            System.out.println(name + ": " + score);
}
```