JC2002 Java Programming

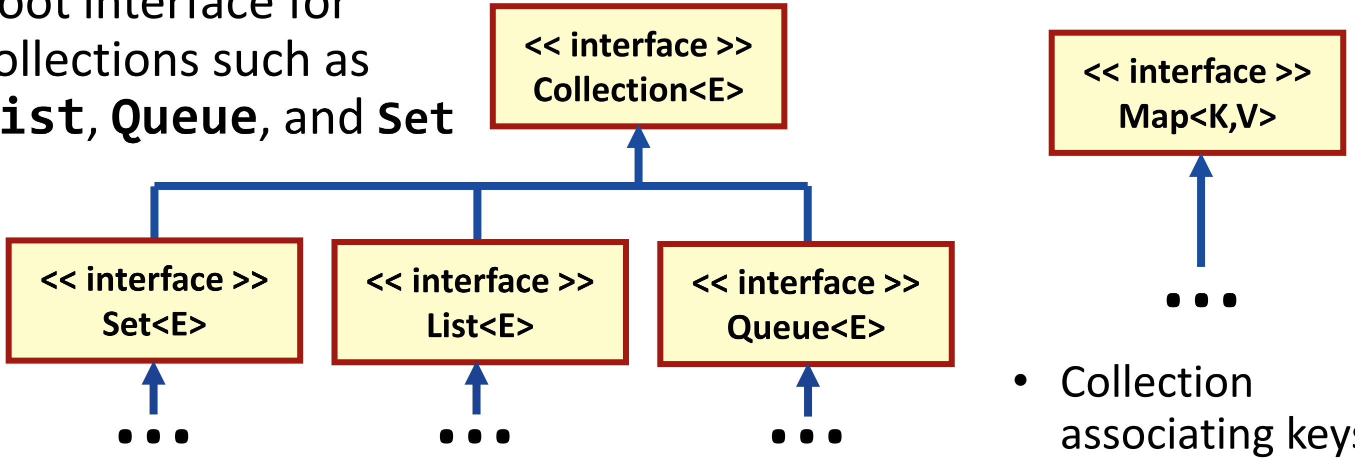
Lecture 34: The basics of collections

Collections

- Any group of individual objects, represented as a single unit, is known as a *collection* of objects
    - For example, a zoo could be defined as a collection of animals
- In Java, a separate framework (*collection framework*) for handling data structures for collections has been defined
    - The main classes and interfaces for collections are included in packages **`java.util.Collection`** and **`java.util.Map`**

Collection interfaces

- Root interface for collections such as **`List`**, **`Queue`**, and **`Set`**



- Collection that ***cannot*** contain duplicates

- Ordered collection that may contain duplicates

- *First-in, first-out* collection modeling *waiting line*

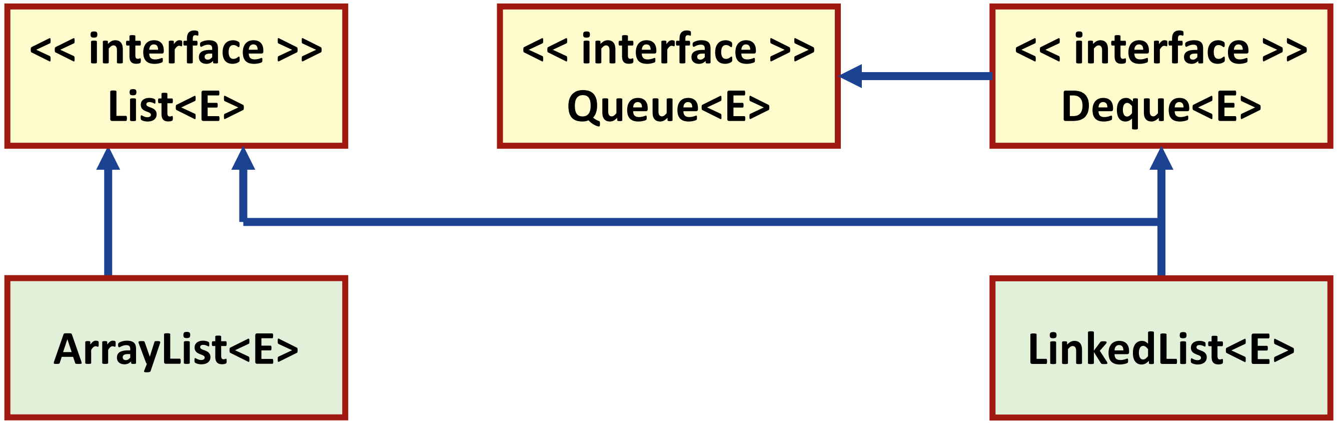
- Collection associating keys to values, cannot contain duplicate keys, not derived from `Collection`!

Collections vs. arrays

- Unlike arrays, collections *can*:
  - Store homogeneous and heterogenous data types
  - Grow in size
  - Perform slower than arrays
- Unlike arrays, collections *cannot*:
  - Store primitive types (*int*, *char* etc.)
- Collections have many support methods defined and are better option when it comes to memory space usage

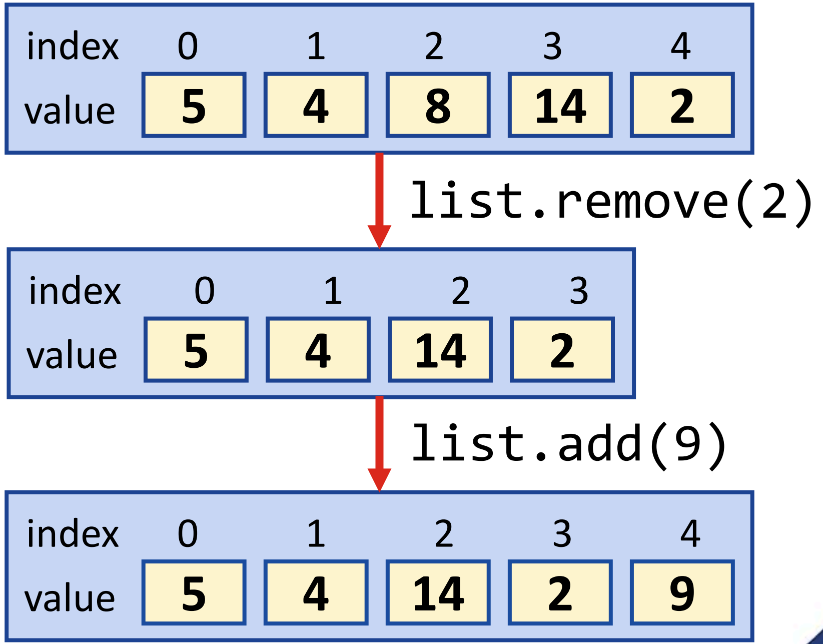
List

- There are two types of list classes: **`ArrayList`** and **`LinkedList`**
- When iterating over `List`, the order of elements is preserved



ArrayList

- Usually, the best choice of list implementations
- Access elements by calling **`list.get(`**`index`**`)`**
- Remove elements by calling **`list.remove(`**`index`**`)`**
- Add elements by calling **`list.add(`**`data`**`)`**



ArrayList example

```java
import java.util.* ;
public class ArrayListExample {
  public static void main ( String[] args) {
    List<String> nameList = new ArrayList<String>();
    System.out.println("initial size: " + nameList.size());
    nameList.add("Bob");      nameList.add("Cecilia");
    nameList.add("Alice");    nameList.add("Daniel");
    System.out.println("new size: " + nameList.size());
    nameList.remove(1);
    System.out.println("new size: " + nameList.size());
    nameList.add("Edward");
    System.out.println("Final name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

We refer to `ArrayList` via `List` interface to allow more flexibility: if we later notice that **`LinkedList`** is more suitable, it is easy to change.

UNIVERSITY OF ABERDEEN

ArrayList example: output

```java
import java.util.* ;
public class ArrayListExample {
  public static void main ( String[] args) {
    List<String> nameList = new ArrayList<String>();
    System.out.println("initial size: " + nameList.size());
    nameList.add("Bob");     nameList.add("Cecilia");
    nameList.add("Alice");   nameList.add("Daniel");
    System.out.println("new size: " + nameList.size());
    nameList.remove(1);
    System.out.println("new size: " + nameList.size());
    nameList.add("Edward");
    System.out.println("Final name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

```
$ java ArrayListExample
initial size: 0
new size: 4
new size: 3
Final name list:
Bob Alice Daniel Edward
$
```

Iterators

- An **`Iterator`** class object can be used to loop through collections
  - “Iterating” is a technical term for looping.
  - Method **`iterator()`** can be used to get an `Iterator` object for any collection.
  - Methods **`hasNext()`** and **`next()`** of the iterator can be used to loop through the collection.
- Note that the **iterator becomes invalid immediately, if the collection is modified using one of its methods**
  - Then, using the iterator throws `ConcurrentModificationException`
  - Helps to avoid two threads to modify collections simultaneously

Iterator example

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");     nameList.add("Cecilia");
    nameList.add("Alice");   nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      // iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

This will NOT work

This will work

UNIVERSITY OF ABERDEEN

Iterator example: output (1)

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");    nameList.add("Cecilia");
    nameList.add("Alice");  nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      // iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

```
$ java IteratorExample
Bob
Cecilia
Alice
Daniel
Size: 4
$
```

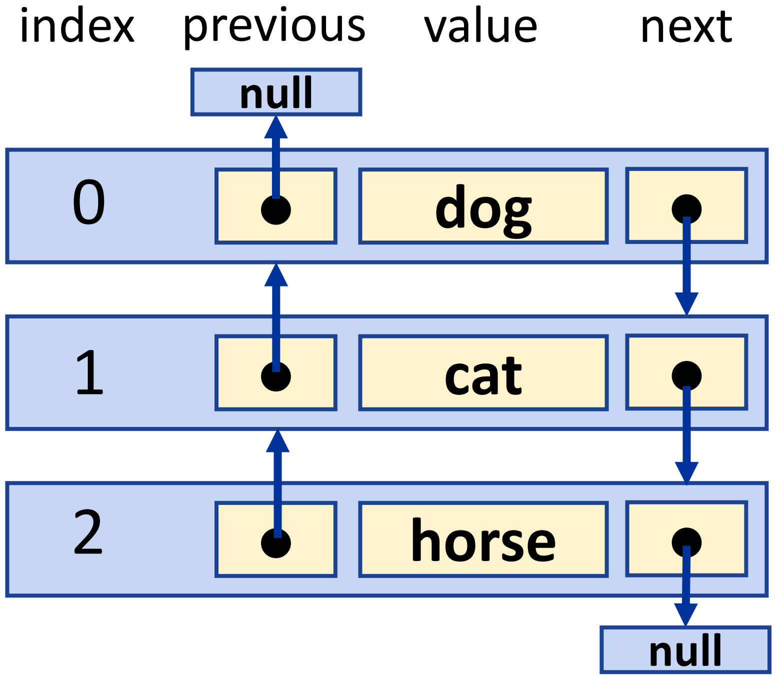
Iterator example: output (2)

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");     nameList.add("Cecilia");
    nameList.add("Alice");   nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

```
$ java IteratorExample
Bob
Cecilia
Alice
Daniel
Size: 0
$
```

LinkedList

- Start with *head*, ends with *tail*
- Elements are linked to the next element
- Efficient insertion and deletion, but bad for memory
- Access elements by calling **`list.get(`**`index`**`)`**
- Add elements by calling **`list.add(`**`data`**`)`** or **`list.push(`**`data`**`)`**



LinkedList example

```
import java.util.* ;
public class LinkedListExample {
  public static void main ( String[] args) {
    LinkedList<String> nameList = new LinkedList<String>();
    nameList.push("Bob");
    nameList.add("Daniel");
    nameList.addFirst("Alice");
    nameList.add(2, "Cecilia");
    System.out.println("Name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

Adds in the beginning of the list

Adds in the end of the list

LinkedList example: output

```java
import java.util.* ;
public class LinkedListExample {
  public static void main ( String[] args) {
    LinkedList<String> nameList = new LinkedList<String>();
    nameList.push("Bob");
    nameList.add("Daniel");
    nameList.addFirst("Alice");
    nameList.add(2, "Cecilia");
    System.out.println("Name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

```
$ java LinkedListExample
Name list:
Alice Bob Cecilia Daniel
$
```

ArrayList vs. LinkedList

- Basically, same things can be done with both types of lists: their main difference is the internal representation of data
    - In `ArrayList`, it is faster to find an element at certain index, because the elements are located in memory in fixed order.
    - In `LinkedList`, it is faster to add and remove elements, because there is no need to move large blocks of data in memory.
- The choice depends on the application
    - `LinkedList` is better if there is often need to add and remove elements
    - `ArrayList` is better if you just need to modify elements without adding or removing

Some important methods in collections

Method	Description
`sort()`	Sorts the elements of a `List`
`binarySearch()`	Locates an element of a `List` using efficient binary search algorithm
`reverse()`	Reverses the elements of a `List`
`shuffle()`	Reorders the elements of a `List` randomly
`fill()`	Sets every element in a `List` to refer to a specific object
`copy()`	Copies references from one `List` to another

Deck of cards example: class Card

```java
import java.util.* ;
class Card {
  public enum Face {Ace, Two, Three, Four, Five, Six, Seven,
                    Eight, Nine, Ten, Jack, Queen, King}
  public enum Suit {Clubs, Diamonds, Spades, Hearts}
  private final Face face;
  private final Suit suit;
  public Card(Face face, Suit suit) {
    this.face = face; this.suit = suit;
  }
  public Face getFace()  { return face; }
  public Suit getSuit()  { return suit; }
  public String toString() {
    return String.format("%s of
  }
}
```

```
$ java LinkedListExample
Name list:
Alice Bob Cecilia Daniel
$
```

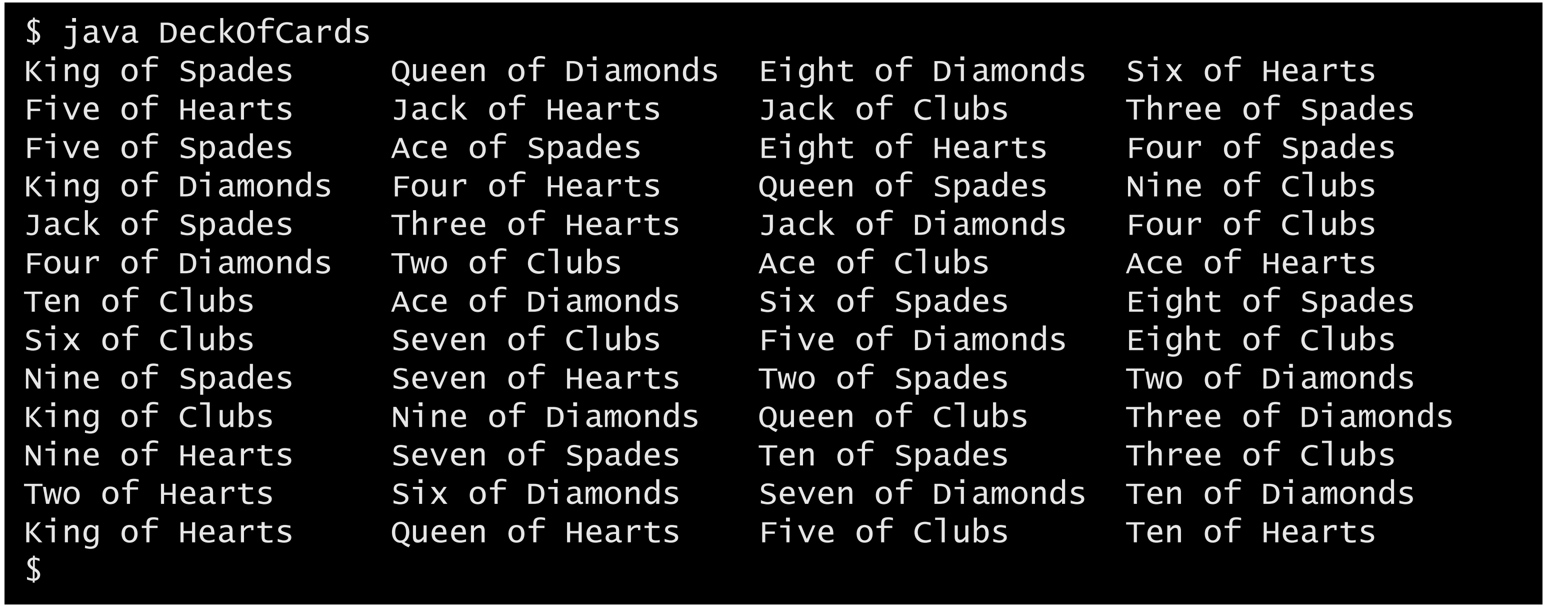
Deck of cards example: using List

```java
public class DeckOfCards {
  private List<Card> cards;
  public DeckOfCards() {
    Card deck[] = new Card[52];
    int count = 0;
    for(Card.Suit suit: Card.Suit.values()) {
      for(Card.Face face: Card.Face.values()) {
        deck[count++] = new Card(face, suit);
      }
    }
    cards = Arrays.asList(deck);
    Collections.shuffle(cards);
  }
  public void printCards() {
    for(int i=0; i<52; i++) {
      System.out.printf("%-19s%s", cards.get(i),
        ((i+1) % 4 == 0) ? "\n" : "");
    }
  }
```

Converts an Array to a List

```java
  public static void main ( String[] args) {
    DeckOfCards deck = new DeckOfCards();
    deck.printCards();
  }
}
```

Deck of cards example: output



```
$ java DeckOfCards
King of Spades     Queen of Diamonds  Eight of Diamonds  Six of Hearts
Five of Hearts     Jack of Hearts     Jack of Clubs      Three of Spades
Five of Spades     Ace of Spades      Eight of Hearts    Four of Spades
King of Diamonds   Four of Hearts     Queen of Spades    Nine of Clubs
Jack of Spades     Three of Hearts    Jack of Diamonds   Four of Clubs
Four of Diamonds   Two of Clubs       Ace of Clubs       Ace of Hearts
Ten of Clubs       Ace of Diamonds    Six of Spades      Eight of Spades
Six of Clubs       Seven of Clubs     Five of Diamonds   Eight of Clubs
Nine of Spades     Seven of Hearts    Two of Spades      Two of Diamonds
King of Clubs      Nine of Diamonds   Queen of Clubs     Three of Diamonds
Nine of Hearts     Seven of Spades    Ten of Spades      Three of Clubs
Two of Hearts      Six of Diamonds    Seven of Diamonds  Ten of Diamonds
King of Hearts     Queen of Hearts    Five of Clubs      Ten of Hearts
$
```

Example of sort() and binarySearch()

```java
import java.util.* ;
public class BinarySearchExample {
  public static void main ( String[] args) {
    String[] names = {"Bob","Alice","Edward","Cecilia","David","Frank"};
    List<String> list = new ArrayList<>(Arrays.asList(names));
    Collections.sort(list);
    for(String name : list) {
      System.out.println(name);
    }
    int index = Collections.binarySearch(list, "Edward");
    System.out.printf("Index of Edward is %d\n", index);
  }
}
```

Output of sort() and binarySearch()

```java
import java.util.* ;
public class BinarySearchExample {
  public static void main ( String[] args) {
    String[] names = {"Bob","Alice","Edward","Cecilia","David","Frank"};
    List<String> list = new ArrayList<>(Arrays.asList(names));
    Collections.sort(list);
    for(String name : list) {
      System.out.println(name);
    }
    int index = Collections.binarySearch(list, "Edward");
    System.out.printf("Index of Edward is %d\n", index);
  }
}
```

```
$ java BinarySearchExample
Alice
Bob
Cecilia
David
Edward
Frank
Index of Edward is 4
$
```

Summary

- In many applications, string operations like searching for substrings, concatenating strings, and modifying string content are essential
  - In Java, `String` objects are immutable: their content cannot be changed after they are created
  - Class `StringBuilder` can be used to create dynamic string buffers
- Regular expressions are commonly used for defining string patterns for searching and validating string content in a specific format
- Java collection framework is useful for handling data structures holding a group of items, such as *lists*, *sets*, and *queues*

Questions, comments?