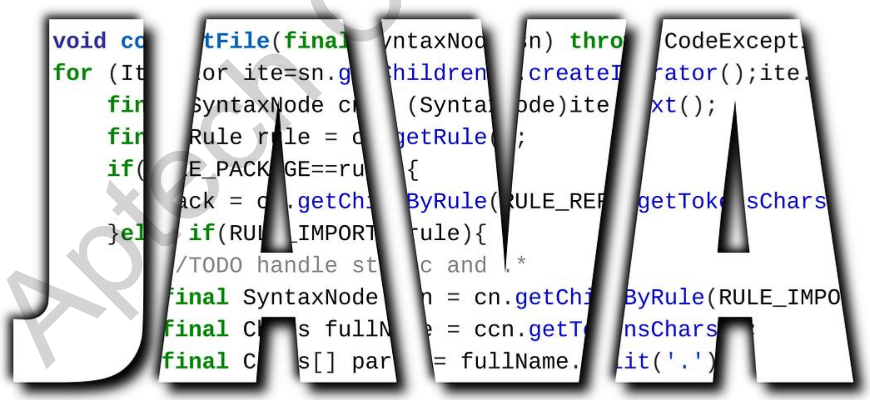
Power Programming with Java

Session: 3

Java Utility APIs

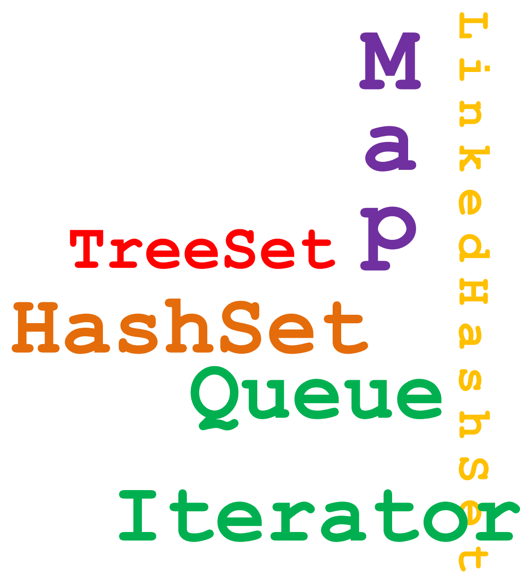
Objectives

- Explain `java.util` package
- Explain List classes and interfaces
- Explain Set classes and interfaces
- Explain Map classes and interfaces
- Explain Queues and Arrays



Introduction

- The Collection framework consist of collection interfaces which are primary means by which collections are manipulated.
- They also have wrapper and general purpose implementations.
- Adapter implementation helps to adapt one collection over other.
- Besides these, there are convenience implementations and legacy implementations.



java.util Package [1-3]

- The java.util package contains the definition of a number of useful classes and interfaces providing a broad range of functionality.
- The package mainly contains collection classes that are useful for working with groups of objects.
- The package also contains the definition of classes that provides date and time facilities.
- It also includes other utilities such as calendar and dictionary.
- It contains a list of classes and interfaces to manage a collection of data in memory.

- Following figure displays some of the classes and interfaces present in java.util package:

java.util Package [2-3]

Date Class, Its Constructors, and Methods:

- The Date class object represents date and time and provides methods for manipulating date and time instances.
- Date is represented as a long type that counts the number of milliseconds since January 1, 1970, 00:00:00 GMT.
- A Date object cannot be printed without converting it to a String type.
- Following table lists the constructors of Date class:

Constructor	Description
Date()	The constructor creates a Date object using today's date.
Date(long dt)	The constructor creates a Date object using specified number of milliseconds since January 1, 1970, 00:00:00 GMT.

Calendar Class, Its Constructors and Methods:

- The Calendar class can retrieve information in the form of integers such as DAY, MONTH, and YEAR based on a given DATE object.
- It is abstract in nature. Therefore, it cannot be instantiated like the Date class.
- A Calendar object provides all the necessary time field values.

Random Class:

- The Random class is used to generate random numbers and generates numbers in an unsystematic or arbitrary manner.
- Random object can be used to simulate a dice throwing game.

Collections Framework

- A collection is a container that helps to group multiple elements into a single unit.
- Collections help to store, retrieve, manipulate, and communicate data.
- The Collections Framework represents and manipulates collections.
- It includes:
  - Algorithms
  - Implementations
  - Interfaces
- Collections Framework consists of interfaces and classes for working with group of objects.

Collection Interface

- At the top of the collection hierarchy lies `Collection` interface, which helps to convert a collection's type.
- This interface is extended by following sub interfaces:
  - `Set`
  - `List`
  - `Queue`
- Some of the `Collection` classes are:
  - `HashSet`
  - `LinkedHashSet`
  - `TreeSet`
- The interface includes following methods:
  - `size`, `isEmpty`: Determine number of elements that exist in the collection.
  - `contains`: Check if a given object is in the collection.
  - `add`, `remove`: Add and remove an element from the collection.
  - `iterator`: Provide an iterator over the collection.
- A few other important methods supported by the `Collection` interface are:

Method	Description
`clear()`	Removes or clears all the contents from the collection
`toArray()`	Returns an array containing all the elements of this collection

Traversing Collections

Using `for-each` construct:

- This helps to traverse a collection or array using a `for` loop.
- Code Snippet illustrates use of the `for-each` construct to print out each element of a collection on a separate line.

Code Snippet

```
for (Object obj : collection)
System.out.println(obj);
```

Using `Iterator`:

- These help to traverse through a collection.
- They also help to remove elements from the collection selectively.
- The `iterator()` method is invoked to obtain an Iterator for a collection.
- The `Iterator` interface includes following methods:

```
public interface Iterator<E> {
      boolean hasNext();
      E next();
      void remove(); //optional
}
```

- Following points explain the `Iterator` interface:
    - The `hasNext()` method returns true if the iteration as more elements.
    - The `next()` method returns the next element in the iteration.
    - The `remove()` method removes the last element which was returned by the `next()` method from the `Collection`.

Bulk Operations

- Bulk operations perform shorthand operations on an entire collection using the basic operations.
- Following table describes the methods for bulk operations:

Method	Description
`containsAll`	This method will return true if the target Collection contains all elements that exist in the specified Collection.
`addAll`	This method will add all the elements of the specified Collection to the target Collection.
`removeAll`	This method will remove all the elements from the target Collection that exist in the specified Collection.
`retainAll`	This method will remove those elements from the target Collection that do not exist in the specified Collection.

Convenience Factory Methods for Collections

- Java had received a lot of criticism for its verbosity until version JDK 8.
- JDK 9 and onwards include Convenience Factory Methods of Collections.
- These enable the users to create small, unmodifiable collection instances with one line of code.
- These also help in creating high-performing and compact collection interface.
- To allow the users to create collections with just a few instances, the Application Programming Interface (API) has been kept minimal.

General Structure of APIs [1-2]

- For creating compact collections and maps, a user can use static factory methods on the Collection interfaces (`Map`, `Set`, and `List`).
- These methods are called `of()`.
- The `of()` method has 11 overloaded versions, each taking zero to 10 elements.
- One overloaded method takes a variable argument (`var-arg`). This creates an immutable collection from an arbitrary number of elements.
- Internally, the method will wrap the constituent elements in an array of that particular kind and pass it.
- JDK 9 onwards provides 11 `of()` methods (taking 0 to 10 elements) for preventing the garbage collection, initialization, and allocation overload.
- The `var-args` overload method must be used to create a collection with more than 10 elements.

General Structure of APIs [2-2]

- Static factory methods added from Java version 9 and onwards to Collection interfaces are:

`List.of()`

- The `of()` method has been added to the `List` interface.

`Set.of()`

- `Set.of()` is similar to the `List.of()` except that it returns a `Set`.

`Map.of()`

- A Map has a set of entries in the form of a key-value pair. The API must facilitate the user to create both.
- The `Map.of()` method has 11 overloaded versions

`Map.ofEntries()` and `Map.entry()`

- The `Map.ofEntries()` accepts a `var-args` argument of type `Map.Entry`.
- It is a nested (inner) interface of the `Map` interface.
- The `Map` interface has added a method, `entry()`, for creating an instance of type `Map.Entry`.
- Calling `setValue()` on the returned Entry throws an `UnsupportedOperationException` since the entry is unmodifiable.

Beware of Nulls

- It is not necessary that all the discussed methods would permit a null.
- This is a good practice because it catches bugs early and prevents runtime exceptions.
- Avoiding nulls enhances runtime performance of the returned Collection.
- A user cannot pass a null to following methods:
  - `Set.of()`
  - `Map.of()`
  - `List.of()`
  - `Map.ofEntries()`
  - `Map.entry()`

Concrete Classes of Factory Methods

- There are concrete type of collections and Map returned by the new convenience factory methods.
- The new methods return an object internal to the JDK.
- These do not belong to the public collection implementations.
- There is no guarantee about the type that is returned and it may also change in the future.
- Hence, a user must program an interface and consider the returned `Object` as a `Map`, `Set`, or `List`.
- A special class is used for `List`, setting not more than two elements and mapping with not more than one entry.

Collection to Array [1-4]

- An array is linear data structure containing elements whose size is defined at the time of creation.
- It can hold primitive homogeneous data or objects.
- A predefined class retaining only heterogeneous object types but primitive is called as a collection.
- A user can use the `List.toArray()` or `List.add()` methods to convent a Collection into arrays.

Approach 1: Using `List.add()` method

- An element E is inserted at a specified position index in the list using `List.add()`.

Syntax

```
public void add (int index, E element);
```

where,

`index` is where the element is to be inserted and

`E` is the element is to be inserted.

- The method may cause `IndexOutOfBoundsException` when the index is not in the range.

Collection to Array [2-4]

Code Snippet illustrates the Java program for changing the Collection of data in list to an array.

Code Snippet

```
. . .
// Or simply add all generic Java libraries
import java.util.*;
public class GFG {
// Main driver method
public static void main(String[] args) {
// Creating arrayList list dynamically
List<String> list = new ArrayList<String>();
// List is created
// Adding elements to the listlist.add("Let's ");
list.add("start ");
list.add("Power ");
list.add("Programming ");
list.add("With ");
list.add("Java ");
// Converting list to an array
String[] str = list.toArray(new String[0]);
// Iterating over elements of array
for (int i = 0; i < str.length; i++) {
String data = str[i];
// Printing elements of an array
System.out.print(data); }}}
```

Collection to Array [3-4]

Approach 2: Using `list.toArray()` method

- This method is present in the `List` interface.
- It returns all the elements of the list in sequential order as an array.

Syntax

```
public Object[] toArray()
```

- Features of `list.toArray()` are:
  - It is determined by `toArray` in interface Collection and interface List.
  - It overrides `toArray` in class `AbstractCollection`
  - It returns an array containing all the elements in this list in the right order.
- Code Snippet explains the `list.toArray()` method and its usage.

Code Snippet

```
// Importing generic Java libraries
import java.util.*;
import java.io.*;
public class GFG {
public static void main(String[] args) {
// Reading input from the user
// via BufferedReader class
```

Collection to Array [4-4]

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
// 'in' is object created of this class
// Creating object of Scanner class
Scanner sc = new Scanner(System.in);
// Creating ArrayList to store user input
List<String> list = new ArrayList<String>();
// Taking input from user
// adding elements to the list
while (sc.hasNext()) {
String i = sc.nextLine();
list.add(i);
}
// Converting list to an array
String[] str = list.toArray(new String[0]);
// Iteration over array
for (int i = 0; i < str.length; i++) {
String data = str[i];
// Printing the elements
System.out.print(data);
}
}
}
```

UnModifiable Collections [1-3]

- The method `unmodifiableCollection()` of `java.util.Collections` class returns an unmodifiable view of the specified Collection.
- It allows different modules to offer read-only access of the internal collections.
- Query operations and any attempt to modify the returned Collection whether directly or via its iterator, will throw an `UnsupportedOperationException`.
- The returned Collection depend on Object's `hashCode()` and `equals()`.
- The returned Collection is `serializable` when the specified Collection is `serializable`.

Syntax

```
public static <T> Collection<T>
unmodifiableCollection(Collection<? extends T> c)
```

- This method takes the collection as a parameter for which an unmodifiable view is to be returned.
- It returns an unmodifiable view of the specified collection.

UnModifiable Collections [2-3]

Following Code Snippets illustrates the `unmodifiableCollection()` method:

Code Snippet – Case A: For `unmodifiableCollection()`

```
// Java program to demonstrate
// unmodifiableCollection() method
// for <Character> Value
import java.util.*;
public class GFG1 {
public static void main(String[]
argv) throws Exception {
try {
// creating object of
ArrayList<Character>
List<Character> list = new
ArrayList<Character>();
// populate the list
list.add('X');
list.add('Y');

//code continues in the right code
block =>
```

```
// printing the list
System.out.println("Initial list:
" + list);
// getting unmodifiable list
// using
//unmodifiableCollection()
//method
Collection<Character>
immutablelist = Collections.
unmodifiableCollection(list);
}
catch
(UnsupportedOperationException e)
{
System.out.println("Exception
thrown : " + e);
}
}
}
```

UnModifiable Collections [3-3]

Code Snippet – Case B: For UnsupportedOperationException

```
// Java program to demonstrate
// unmodifiableCollection() method
// for
//UnsupportedOperationException
import java.util.*;
public class GFG1 {
public static void main(String[]
argv) throws Exception {
try {
// creating object of
ArrayList<Character>
List<Character> list = new
ArrayList<Character>();
// populate the list
list.add('X');
list.add('Y');
// printing the list
System.out.println("Initial list: "
+ list);
//code continues in right code
//block=>
```

```
// getting unmodifiable list
// using unmodifiableCollection()
//method
Collection<Character> immutablelist =
Collections.unmodifiableCollection(list);
// Adding element to new Collection
System.out.println("\nTrying to modify"+
" the unmodifiableCollection");
immutablelist.add('Z');
}
catch (UnsupportedOperationException e) {
System.out.println("Exception thrown : "
+ e);
}
}
}
```

Lists

- The `List` interface is an extension of the `Collection` interface.
- It defines an ordered collection of data.
- It allows duplicate objects to be added to a list.
- It adds position-oriented operations.
- It enables programmers to work with a part of the list.
- The `List` interface uses an index for ordering the elements while storing them in a list.
- `List` has methods that allow access to elements based on their position.
- The methods can:
  - search for a specific element
  - return their position
  - Perform arbitrary range operations and more

Methods of `List` Interface

`add(int index, E element)`

`addAll(int index, Collection<? extends E> c)`

`get(int index)`

`set(int index, E element)`

Methods of `List` Interface

`subList(int start, int end)`

`indexOf(Object o)`

`remove(int index)`

`lastIndexOf(Object o)`

ArrayList Class [1-2]

- Is an implementation of the List interface in the Collections Framework.
- Creates a variable-length array of object references.
- Includes all elements, including null.
- Provides methods to change the size of the array that is used internally to store the list.
- Each instance of the class includes a capacity representing the size of the array.
- A capacity stores the elements in the list and grows automatically as elements are added to an ArrayList.

ArrayList Class [2-2]

- An instance of `ArrayList` can be created using any one of following constructors:
  - `ArrayList()`
  - `ArrayList(Collection <? extends E> c)`
  - `ArrayList(int initialCapacity)`
- In the Code Snippet, the creation of an instance of the `ArrayList` class is displayed.

Code Snippet

```
  . . .
List<String> listObj = new ArrayList<String> ();
System.out.println("The size is : " + listObj.size());
for (int ctr=1; ctr <= 10; ctr++)
{
     listObj.add("Value is : " + new Integer(ctr));
}
. . .
```

Methods of `ArrayList` Class

- `add(E obj)`
- `trimToSize()`
- `ensureCapacity(int minCap)`
- `clear()`
- `contains(Object obj)`
- `size()`

Code Snippet displays the use of `ArrayList` class.

Code Snippet

```
  . . .
List<String> listObj = new ArrayList<String> ();
System.out.println("The size is : " + listObj.size());
for (int ctr=1; ctr <= 10; ctr++)
{
     listObj.add("Value is : " + new Integer(ctr));
}
listObj.set(5, "Hello World");
System.out.println("Value is: " +(String)listObj.get(5));
. . .
```

Vector Class [1-2]

- The `Vector` class is similar to an `ArrayList` as it also implements dynamic array.
- It stores an array of objects.
- The size of the array can increase or decrease.
- The elements in `Vector` can be accessed using an integer index.
- Each vector maintains a capacity and a `capacityIncrement` to optimize storage management.
- In the Code Snippet, the creation of an instance of the `Vector` class is displayed.

Code Snippet

```
. . .
Vector vecObj = new Vector();
. . .
```

Vector Class [2-2]

Methods of `Vector` Class:

- `addElement(E obj)`
- `capacity()`
- `toArray()`
- `elementAt(int pos)`
- `removeElement(Object obj)`
- `clear()`

- Code Snippet displays the use of the `Vector` class.

Code Snippet

```
. . .
Vector<Object> vecObj = new Vector<Object>();
vecObj.addElement(new Integer(5));
vecObj.addElement(new Integer(7));
vecObj.addElement(new Integer(45));
vecObj.addElement(new Float(9.95));
vecObj.addElement(new Float(6.085));
System.out.println("The value is: " +(Object)vecObj.elementAt(3));
. . .
```

LinkedList Class

- `LinkedList` class implements the `List` interface.
- A linked list is a list of objects having a link to the next object.
- Linked lists allow insertion and removal of nodes at any position in the list.
- These lists do not allow random access.
- Different types of linked lists: singly-linked lists, doubly-linked lists, and circularly-linked lists.
- Java provides the `LinkedList` class in the `java.util` package to implement linked lists.
- **`LinkedList()`** constructor creates an empty linked list.

`LinkedList(Collection <? extends E>c):`

- The `LinkedList(Collection <? extends E>c)` constructor creates a linked list.
- It contains the elements of a specified collection.

In the Code Snippet, the creation of an instance of the `LinkedList` class is displayed.

Code Snippet

```
. . .
LinkedList<String> lisObj = new LinkedList<List>();
. . .
```

Methods of `LinkedList` Class

In the Code Snippet, the use of the methods of the `LinkedList` class is displayed.

Code Snippet

```
. . .
LinkedList<String> lisObj = new LinkedList<String>();
lisObj.add("John");
lisObj.add("Mary");
lisObj.add("Jack");
lisObj.add("Elvis");
lisObj.add("Martin");
System.out.println("Original content of the list: " +
lisObj);
lisObj.removeFirst();
System.out.println("After removing content of the list: " +
lisObj);
. . .
```

AutoBoxing and Unboxing

- The autoboxing and unboxing feature automates the process of using primitive value into a collection.
- Collections hold only object references.
- Primitive values such as `int` from `Integer`, have to be boxed into the appropriate wrapper class.
- If an `int` value is required, the integer value must be unbox using the `intValue()` method.
- The autoboxing and unboxing feature helps to reduce the clutter in the code.

Sets

- The `Set` interface creates a list of unordered objects.
- It creates non-duplicate list of object references.
- The `Set` interface inherits all the methods from the `Collection` interface, except those allowing duplicate elements.
- The Java platform contains three general-purpose `Set` implementations. They are:

- The `Set` interface is an extension of the `Collection` interface
- It defines a set of elements.
- The difference between `List` and `Set` is that, the `Set` does not permit duplication of elements.
- Therefore, `add()` method returns false if duplicate elements are added.

Methods of Set Interface

`containsAll(Collection<?> obj)`

`addAll(Collection<? extends E> obj)`

`retainAll(Collection<?> obj)`

`removeAll(Collection<?> obj)`

SortedSet Interface

- The `SortedSet` interface extends the `Set` interface.
- Its iterator traverses its elements in the ascending order.
- `SortedSet` is used to create sorted lists of non-duplicate object references.
- The ordering of a sorted set should be consistent with `equals()` method.
- A sorted set performs all element comparisons using the `compareTo()` or `compare()` method.
- Typically, sorted set implementation classes provide following standard constructors:
  - `No argument (void) constructor`
  - `Single argument of type Comparator constructor`
  - `Single argument of type Collection constructor`
  - `Single argument of type SortedSet constructor`

HashSet Class

- `HashSet` class implements the `Set` interface.
- It creates a collection that makes use of a hashtable for data storage.
- This `HashSet` class allows null element.
- The `HashSet` class provides constant time performance for the basic operations.
- In the Code Snippet, the creation of an instance of `HashSet` class is displayed.

Code Snippet

```
. . .
Set<String> words = new HashSet<String>();
. . .
```

LinkedHashSet Class

- The `LinkedHashSet` class creates a list of elements.
- It maintains the order of the elements added to the `Set`.
- This class includes following features:
  - It provides all of the optional `Set` operations.
  - It permits null elements.
  - It provides constant-time performance for the basic operations such as `add` and `remove`.
- The constructors of this class are:
  - `LinkedHashSet()`
  - `LinkedHashSet(Collection<? extends E> c)`
  - `LinkedHashSet(int initial capacity)`

TreeSet Class

- TreeSet class implements the NavigableSet interface.
- It uses a tree structure for data storage.
- The elements can be ordered by natural ordering.
- A user can also use a Comparator provided at the time of Set creation.
- Objects are stored in ascending order.
- TreeSet is used when elements have to be extracted quickly from the collection in a sorted manner.
- In the Code Snippet, an instance of TreeSet is created.

Code Snippet

```
. . .
TreeSet tsObj = new TreeSet();
. . .
```

Maps

- A `Map` object stores data in the form of relationships between keys and values.
- Each key will map to at least a single value.
- If key information is known, its value can be retrieved from the `Map` object.
- Keys should be unique but values can be duplicated.
- The `Map` interface does not extend the `Collection` interface.
- The Collections API provides three general-purpose `Map` implementations:
  - `HashMap`
  - `TreeMap`
  - `LinkedHashMap`

HashMap Class [1-3]

- The `HashMap` class implements the `Map` interface and inherits all its methods.
- An instance of `HashMap` has two parameters:
  - Initial capacity
  - Load factor.
- Initial capacity determines the number of objects that can be added to the `HashMap` at the time of the `Hashtable` creation.
- The load factor determines how full the `Hashtable` can get, before its capacity is automatically increased.

HashMap Class [2-3]

Code Snippet displays the use of the `HashMap` class.

Code Snippet

```
. . .
class EmployeeData {
      public EmployeeData(String nm){
            name = nm;
            salary = 5600;
      }
      public String toString() {
      return "[name=" + name + ",
salary=" + salary + "]";
           public String toString()
      {
      return "[name=" + name + ",
salary=" + salary + "]";
      }
      . . .
      }
      //code continues on right
      //block =>
```

```
public class MapTest {
   public static void main(String[] args) {
      Map<String, EmployeeData> staffObj =
new HashMap<String, EmployeeData>();
      staffObj.put("101", new
EmployeeData("Anna John"));
      staffObj.put("102", new
EmployeeData("Harry Hacker"));
      staffObj.put("103", new
EmployeeData("Joby Martin"));
      System.out.println(staffObj);
      staffObj.remove("103");
      staffObj.put("106", new
EmployeeData("Joby Martin"));
      System.out.println(staffObj.get("106")
);
      System.out.println(staffObj);
      . . .
      }
}
```

Hashtable Class

- The `Hashtable` class implements the `Map` interface.
- However, it stores elements as a key/value pairs in the `hashtable`.
- While using a `Hashtable`, a key is specified to which a value is linked.
- The class inherits all the methods of the `Map` interface.
- To retrieve and store objects from a `hashtable` successfully, objects used as keys must implement `hashCode()` and `equals()` methods.

Code Snippet displays the use of the `Hashtable` class.

Code Snippet

```
. . .
Hashtable<String, String> bookHash = new Hashtable<String, String>();
bookHash.put("115-355N", "A Guide to Advanced Java");
bookHash.put("116-455A", "Learn Java by Example");
bookHash.put("116-466B", "Introduction to Solaris");
String str = (String) bookHash.get("116-455A");
System.out.println("Detail of a book " + str);
System.out.println("Is table empty " + bookHash.isEmpty());
System.out.println("Does table contains key? " + bookHash.containsKey("116- 466B"));
Enumeration name = bookHash.keys();
while (name.hasMoreElements()) {
        String bkCode = (String)name.nextElement();
        System.out.println( bkCode +": " + (String)bookHash.get(bkCode));
}
. . .
```

TreeMap Class

- The `TreeMap` class implements the `NavigableMap` interface and stores elements in a tree structure.
- The `TreeMap` returns keys in sorted order.
- If there is no requirement to retrieve `Map` elements sorted by key, then `HashMap` would be a more practical structure to use.
- Important methods of the `TreeMap` class are:
  - `firstKey()`
  - `lastKey()`
  - `headMap(K toKey)`
  - `tailMap(K fromKey)`
- Code Snippet displays the use of the `TreeMap` class.

Code Snippet

```
. . .
TreeMap<String, EmployeeData> staffObj = new TreeMap<String, EmployeeData>();
staffObj.put("101", new EmployeeData("Anna John"));
staffObj.put("102", new EmployeeData("Harry Hacker"));
staffObj.put("103", new EmployeeData("Joby Martin"));
System.out.println(staffObj);
staffObj.remove("103");
staffObj.put("104", new EmployeeData("John Luther"));
System.out.println(staffObj.get("104"));
Object firstKey = staffObj.firstKey();
System.out.println(firstKey.toString());
System.out.println((String)staffObj.firstKey());
System.out.println((String)(staffObj.lastKey()));
. . .
```

LinkedHashMap Class

- `LinkedHashMap` class implements the concept of `hashtable` and the linked list in the `Map` interface.
- A `LinkedHashMap` maintains the values in the order they were inserted.
- The important methods in `LinkedHashMap` class are:
  - `clear()`
  - `containsValue(Object value)`
  - `get(Object key)`
  - `removeEldestEntry(Map.Entry<K,V> eldest)`

Stack and Queues

- In the `Stack` class, the stack of objects results in a Last-In-First-Out (LIFO) behavior.
- It extends the `Vector` class to consider a vector as a stack.
- Stack only defines the default constructor that creates an empty stack.
- It includes all the methods of the `vector` class.
- This interface includes following five methods:
  - `empty()`
  - `peek()`
  - `pop()`
  - `push(E item)`
  - `int search(Object o)`

Queue Interface

- A `Queue` is a collection for holding elements that must be processed.
- In `Queue`, the elements are normally ordered in First-In-First-Out (FIFO) manner.
- A queue can be arranged in other orders too.
- Every `Queue` implementation defines ordering properties.
- In a FIFO queue, new elements are inserted at the end of the queue.
- LIFO queues or stacks order the elements in LIFO pattern.
- However, in any form of ordering, a call to the `poll()` method removes the head of the queue.

Deque Interface

- A double ended queue is commonly called `deque`.
- It is a linear collection that supports insertion and removal of elements from both ends.
- `Deque` implementations have no restrictions on the number of elements to include.
- A `deque` when used as a queue results in FIFO behavior.
- When used with the `Stack` class, it provides a consistent set of LIFO stack operations.
- Code Snippet displays `Deque`.

Code Snippet

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

PriorityQueue Class

- Priority queues are similar to queues.
- Their elements are arranged in a user-defined manner and ordered either by natural ordering or according to a comparator.
- A priority queue neither allows adding of non-comparable objects nor allows null elements.
- It is unbound.
- When the elements are added to a priority queue, its capacity grows automatically.
- The `PriorityQueue` class inherits the method of the `Queue` class. Following Code Snippet displays the use of the `PriorityQueue` class:

Code Snippet

```
. . .
PriorityQueue<String> queue = new PriorityQueue<String>();
queue.offer("New York");
queue.offer("Kansas");
queue.offer("California");
queue.offer("Alabama");
System.out.println("1. " + queue.poll()); // removes
System.out.println("2. " + queue.poll()); // removes
System.out.println("3. " + queue.peek());
System.out.println("4. " + queue.peek());
System.out.println("5. " + queue.remove()); // removes
System.out.println("6. " + queue.remove()); // removes
System.out.println("7. " + queue.peek());
System.out.println("8. " + queue.element());// Throws Exception
. . .
```

Arrays Class

- `Arrays` class provides a number of methods for working with arrays such as:
  - Searching
  - Sorting
  - Comparing arrays
- The class has a static factory method allowing the array to be viewed as lists.
- The methods of this class throw an exception, `NullPointerException` if the array reference is null.
- Some of the important methods of this class are:
  - `equals(<type> arrObj1, <type> arrObj2)`
  - `fill(<type>[] array, <type> value)`
  - `fill(type[] array, int fromIndex, int toIndex, type value)`
  - `sort(<type>[] array)`
  - `sort(<type> [] array, int startIndex, int endIndex)`
  - `toString(<type>[] array)`

Sorting Collections

Collection API provides following two interfaces for ordering interfaces:

- **`Comparable`**:
  - The `Comparable` interface imposes a total ordering on the objects of each class which implements it.
  - Lists of objects implementing this interface are automatically sorted.
  - It is sorted using **`Collection.sort`** or **`Arrays.sort`** method.
- **`Comparator`**:
  - This interface provides multiple sorting options.
  - It imposes a total ordering on some collection of objects.

Enhancements in Collection Classes [1-8]

- The `ArrayDeque` class implements the `Deque` interface.
- This class is faster than stack and linked list when used as a queue.
- In the Code Snippet, the use of some of the methods available in the `ArrayDeque` class is displayed.

Code Snippet

```
import java.util.ArrayDeque;
import java.util.Iterator;
...
...
public static void main(String args[]) {
            ArrayDeque arrDeque = new ArrayDeque();
            arrDeque.addLast("Mango"); arrDeque.addLast("Apple");
            arrDeque.addFirst("Banana");
            for (Iterator iter = arrDeque.iterator(); iter.hasNext();) {
            System.out.println(iter.next());
     }
     for (Iterator descendingIter = arrDeque.descendingIterator();
descendingIter.hasNext();) {
              System.out.println(descendingIter.next());
     }
     System.out.println("First Element : " + arrDeque.getFirst());
     System.out.println("Last Element : " + arrDeque.getLast());
     System.out.println("Contains \"Apple\" : " + arrDeque. contains("Apple"));
}
...
```

Enhancements in Collection Classes [2-8]

- The `ConcurrentSkipListSet` class implements the `NavigableSet` interface.
- The `Comparator` is an interface that uses the `compare()` method to sort objects that don't have a natural ordering.
- Code Snippet shows the use of some of the methods available in `ConcurrentSkipListSet` class.

Code Snippet

```
import java.util.Iterator;
import java.util.concurrent.ConcurrentSkipListSet;
...
public static void main(String args[]) {
    ConcurrentSkipListSet fruitSet = new ConcurrentSkipListSet();
    fruitSet.add("Banana");
    fruitSet.add("Peach");
    fruitSet.add("Apple");
    fruitSet.add("Mango");
    fruitSet.add("Orange");
    // Displays in ascending order
    Iterator iterator = fruitSet.iterator();
    System.out.print("In ascending order :");
    while (iterator.hasNext())
    System.out.print(iterator.next() + " ");
    // Displays in descending order
```

Enhancements in Collection Classes [3-8]

```
    System.out.println("In descending order: " +
    fruitSet.descendingSet() + "\n");
    System.out.println("Lower element: " + fruitSet.lower("Mango"));
    System.out.println("Higher element: " + fruitSet.higher("Apple"));
}
...
```

- The `ConcurrentSkipListMap` class executes `ConcurrentNavigableMap` interface.
- Like `ConcurrentHashMap` class, the `ConcurrentSkipListMap` class allows modification without locking the entire map.
- In the Code Snippet, the use of some of the methods available in `ConcurrentSkipListMap` class is displayed.

Code Snippet

```
import java.util.concurrent.ConcurrentSkipListMap;
...
...
public static void main(String args[]) {
ConcurrentSkipListMap fruits = new ConcurrentSkipListMap();
fruits.put(1, "Apple");
fruits.put(2, "Banana");
fruits.put(3, "Mango");
fruits.put(4, "Orange");
```

Enhancements in Collection Classes [4-8]

```
fruits.put(5, "Peach");
// Retrieves first data
System.out.println("First data: " + fruits.firstEntry() + "\n");
// Retrieves last data
System.out.println("Last data: " + fruits.lastEntry() + "\n");
// Displays all data in descending order
System.out.println("Data in reverse order: " + fruits.descendingMap());
}
...
```

- The `LinkedBlockingDeque` class implements the `BlockingDeque` interface.
- The class belongs to `java.util.concurrent` package.
- In the Code Snippet, the implementation of `LinkedBlockingDeque` class and use of some of its available methods is displayed.

Code Snippet

```
/* ProducerDeque.Java */
import java.util.concurrent.BlockingDeque;
class ProducerDeque implements Runnable {
private String name;
private BlockingDeque blockDeque;
public ProducerDeque(String name, BlockingDeque blockDeque) {
this.name = name;
this.blockDeque = blockDeque;
}
```

Enhancements in Collection Classes [5-8]

```
public void run() { for (int i = 1; i < 10; i++) {
         try {
              blockDeque.addFirst(i);
              System.out.println(name + " puts " + i);
              Thread.sleep(100);
          } catch (InterruptedException e) {
          e.printStackTrace();
          } catch (IllegalStateException ex) {
          System.out.println("Deque filled upto the maximum capacity");
          System.exit(0);
          } }
```

- The `LinkedBlockingDeque` class implements the `BlockingDeque` interface.
- The class belongs to `java.util.concurrent` package.
- The class contains linked nodes that are dynamically created after each insertion.
- Following Code Snippet shows the implementation of `LinkedBlockingDeque` class:

Code Snippet

```
/* ProducerDeque.Java */
import java.util.concurrent.BlockingDeque;
class ProducerDeque implements Runnable {
         private String name;
         private BlockingDeque blockDeque;
         public ProducerDeque(String name, BlockingDeque blockDeque) {
         this.name = name;
```

Enhancements in Collection Classes [6-8]

```
public ProducerDeque(String
name, BlockingDeque blockDeque) {
this.name = name;
this.blockDeque =
blockDeque;
}
public void run() {
for (int i = 1; i < 10; i++) {
try {
      blockDeque.addFirst(i);
      System.out.println(name + "
      puts " + i);
      Thread.sleep(100);
     } catch (InterruptedException e) {
     e.printStackTrace();
     } catch (IllegalStateException ex){
     System.out.println("Deque filled
     upto the maximum capacity");
     System.exit(0);
     }
     }
     //code continues on right code
     //block =>
```

```
/* ConsumerDeque.Java */
import java.util.concurrent.BlockingDeque;
import
java.util.concurrent.LinkedBlockingDeque;
class ConsumerDeque implements Runnable {
private String name;
private BlockingDeque blockDeque;
public ConsumerDeque(String name,
BlockingDeque blockDeque) {
this.name = name;
this.blockDeque = blockDeque;
public void run() {
for (int i = 1; i < 10; i++) {
try {
int j = (Integer) blockDeque.peekFirst();
System.out.println(name + " takes " + j);
Thread.sleep(100);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}
```

Enhancements in Collection Classes [7-8]

- The `LinkedBlockingDeque` class implements the `BlockingDeque` interface.
- The class belongs to `java.util.concurrent` package.
- In the Code Snippet, the implementation of `LinkedBlockingDeque` class and use of some of its available methods is displayed.

Code Snippet

```
/* LinkedBlockingDequeClass.Java */
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;
public class LinkedBlockingDequeClass {
public static void main(String[] args) {
BlockingDeque blockDeque = new LinkedBlockingDeque(5);
Runnable produce = new ProducerDeque("Producer", blockDeque);
Runnable consume = new ConsumerDeque("Consumer", blockDeque);
new Thread(produce).start();
new Thread(consume).start();
}
}
```

Enhancements in Collection Classes [8-8]

- The `AbstractMap.SimpleEntry` is static class nested inside `AbstractMap` class.
- The `getKey()` method returns the key of an entry in the instance.
- In the Code Snippet, the implementation of `AbstractMap.SimpleEntry` static class and the use of some of its available methods is displayed.

Code Snippet

```
AbstractMap.SimpleEntry<String,String> se = new
AbstractMap.SimpleEntry<String,String>("1","Apple");
System.out.println(se.getKey());
System.out.println(se.getValue());
se.setValue("Orange");
System.out.println(se.getValue());
```

- The `AbstractMap.SimpleImmutableEntry` class is a static class.
- It is nested inside the `AbstractMap` class.
- If any attempt to change a value is made, it results in throwing `UnsupportedOperationException`.

Summary

- The `java.util` package contains the definition of number of useful classes providing a broad range of functionality.
- The `List` interface is an extension of the Collection interface.
- The `Set` interface creates a list of unordered objects.
- A `Map` object stores data in the form of relationships between keys and values.
- A `Queue` is a collection for holding elements before processing.
- `ArrayDeque` class does not put any restriction on capacity and does not allow null values.
- `AbstractMap.SimpleEntry` is used for implementation of custom map.
- `AbstractMap.SimpleImmutableEntry` class is a static class and does not allow modification of values in an entry.