Unit 9: Understanding Core Packages in Java [3 Hrs]

1. Using java.lang Package

The `java.lang` package is automatically imported into every Java program. It provides fundamental classes and methods that are essential for Java programming.

a. java.lang.Math The `Math` class provides methods for performing basic numeric operations such as exponential, logarithm, square root, and trigonometric functions.

Lab 1: java.lang.Math

```
public class MathExample {
    public static void main(String[] args) {
        double num1 = 25.0;
        double num2 = 4.0;

        System.out.println("Square root of " + num1 + " is: " + Math.sqrt(num1));
        System.out.println("Power of " + num1 + " raised to " + num2 + " is: " +
Math.pow(num1, num2));
        System.out.println("Maximum between " + num1 + " and " + num2 + " is: " +
Math.max(num1, num2));
        System.out.println("Minimum between " + num1 + " and " + num2 + " is: " +
Math.min(num1, num2));
    }
}
```

Sample Output:

```
Square root of 25.0 is: 5.0
Power of 25.0 raised to 4.0 is: 390625.0
Maximum between 25.0 and 4.0 is: 25.0
Minimum between 25.0 and 4.0 is: 4.0
```

Explanation:

- `Math.sqrt()` calculates the square root of a number.
- `Math.pow()` raises a number to the power of another.
- `Math.max()` returns the greater of two numbers.
- `Math.min()` returns the smaller of two numbers.

Commonly used methods in the `Math` class

1. **`Math.abs(double a)`** : Returns the absolute value of a number.
2. **`Math.sqrt(double a)`** : Returns the square root of a number.
3. **`Math.cbrt(double a)`** : Returns the cube root of a number.
4. **`Math.pow(double a, double b)`** : Returns the value of the first argument raised to the power of the second argument.
5. **`Math.exp(double a)`** : Returns Euler's number (e) raised to the power of a number.
6. **`Math.log(double a)`** : Returns the natural logarithm (base e) of a number.
7. **`Math.log10(double a)`** : Returns the base 10 logarithm of a number.
8. **`Math.sin(double a)`** : Returns the trigonometric sine of an angle (in radians).
9. **`Math.cos(double a)`** : Returns the trigonometric cosine of an angle (in radians).

10. **`Math.tan(double a)`** : Returns the trigonometric tangent of an angle (in radians).
11. **`Math.round(double a)`** : Returns the closest integer to the argument, as a `long` or `int` .
12. **`Math.ceil(double a)`** : Returns the smallest integer greater than or equal to the argument.
13. **`Math.floor(double a)`** : Returns the largest integer less than or equal to the argument.

b. Wrapper Classes Wrapper classes provide a way to use primitive data types (`int` , `char` , `boolean` , etc.) as objects. Each primitive type has a corresponding wrapper class.

Lab 2: Wrapper Classes

```
public class WrapperExample {
    public static void main(String[] args) {
        Integer intObj = Integer.valueOf(10);    //Integer intObj = 10; // Autoboxing: Java automatically wraps the primitive into an object
        Double doubleObj = Double.valueOf(5.5);
        Character charObj = Character.valueOf('A');
        Boolean boolObj = Boolean.valueOf(true);

        System.out.println("Integer Object: " + intObj);
        System.out.println("Double Object: " + doubleObj);
        System.out.println("Character Object: " + charObj);
        System.out.println("Boolean Object: " + boolObj);

        int intValue = intObj.intValue();
        double doubleValue = doubleObj.doubleValue();
        char charValue = charObj.charValue();
        boolean boolValue = boolObj.booleanValue();

        System.out.println("Primitive int: " + intValue);
        System.out.println("Primitive double: " + doubleValue);
        System.out.println("Primitive char: " + charValue);
        System.out.println("Primitive boolean: " + boolValue);
    }
}
```

Sample Output:

```
Integer Object: 10
Double Object: 5.5
Character Object: A
Boolean Object: true
Primitive int: 10
Primitive double: 5.5
Primitive char: A
Primitive boolean: true
```

Explanation:

- Wrapper classes like `Integer`, `Double`, `Character`, and `Boolean` are used to convert primitive types into objects by using the `valueOf()` method or Autoboxing (Automatic Conversion).
- Methods like `intValue()`, `doubleValue()`, `charValue()`, and `booleanValue()` are used to retrieve the primitive value from the wrapper object.

Difference Between Primitive Types and Wrapper Classes:

- Primitive types are not objects and are stored directly in the stack memory, whereas wrapper classes are objects stored in the heap memory and offer additional methods for manipulation.

Additional methods provided by wrapper classes for manipulation:

1. **`Integer.parseInt(String s)`**: Converts a string representation of an integer into its primitive `int` value.
2. **`Integer.valueOf(String s)`**: Converts a string representation of an integer into an `Integer` object.
3. **`Double.parseDouble(String s)`**: Converts a string representation of a floating-point number into its primitive `double` value.
4. **`Double.valueOf(String s)`**: Converts a string representation of a floating-point number into a `Double` object.
5. **`Integer.toString(int i)`**: Converts an integer into its string representation.
6. **`Double.toString(double d)`**: Converts a double into its string representation.
7. **`Character.isLetter(char c)`**: Checks if the specified character is a letter (e.g., 'A', 'b').
8. **`Character.isDigit(char c)`**: Checks if the specified character is a digit (e.g., '0', '9').
9. **`Boolean.booleanValue()`**: Returns the primitive `boolean` value of a `Boolean` object.
10. **`Integer.compareTo(Integer another)`**: Compares two `Integer` objects numerically and returns 0, >0, or <0 based on equality or order.
11. **`Double.compare(double d1, double d2)`**: Compares two `double` values numerically and returns 0, >0, or <0 based on equality or order.
12. **`Character.toLowerCase(char c)`**: Converts the given character to its lowercase equivalent.
13. **`Character.toUpperCase(char c)`**: Converts the given character to its uppercase equivalent.

---

2. Using java.util Package

The `java.util` package contains utility classes and interfaces, such as collections, date and time facilities, and random number generation.

Core Classes

- **Vector**: A dynamic array that can grow or shrink in size.
- **Stack**: A last-in-first-out (LIFO) data structure.
- **Dictionary**: An abstract class that maps keys to values (deprecated in favor of `Map`).
- **Hashtable**: A synchronized implementation of a hash table.
- **Enumerations**: An interface for iterating through a collection of elements.
- **Random Number Generation**: The `Random` class is used to generate random numbers.

Lab 3: Vector and Stack:

```java
import java.util.Vector;
import java.util.Stack;

public class VectorStackExample {
    public static void main(String[] args) {
        // Vector Example
        Vector<String> vector = new Vector<>();
        vector.add("Apple");
        vector.add("Banana");
        vector.add("Cherry");
        System.out.println("Vector: " + vector);

        // Stack Example
        Stack<String> stack = new Stack<>();
        stack.push("Red");
        stack.push("Green");
        stack.push("Blue");
        System.out.println("Stack: " + stack);
        System.out.println("Popped Element: " + stack.pop());
        System.out.println("Stack after pop: " + stack);
    }
}
```

Sample Output:

```
Vector: [Apple, Banana, Cherry]
Stack: [Red, Green, Blue]
Popped Element: Blue
Stack after pop: [Red, Green]
```

Explanation:

- `Vector` is a resizable array that can store elements.
- `Stack` is a LIFO data structure where elements are added and removed from the top.
- `push()` adds an element to the stack, and `pop()` removes the top element.

Methods in the `Vector` class:

1. **`add(E element)`**: Appends the specified element to the end of the vector.
2. **`get(int index)`**: Returns the element at the specified position in the vector.
3. **`remove(Object o)`**: Removes the first occurrence of the specified element from the vector.
4. **`remove(int index)`**: Removes the element at the specified position from the vector.
5. **`size()`**: Returns the number of elements currently in the vector.
6. **`contains(Object o)`**: Checks if the vector contains the specified element, returning `true` if found.
7. **`indexOf(Object o)`**: Returns the index of the first occurrence of the specified element, or `-1` if not found.
8. **`set(int index, E element)`**: Replaces the element at the specified position with the given element.
9. **`clear()`**: Removes all elements from the vector, making it empty.

10. **`addElement(E obj)`** : Adds the specified element to the end of the vector (legacy method).

Lab 4: Hashtable and Enumerations:

```
import java.util.Hashtable;
import java.util.Enumeration;

public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, Integer> hashtable = new Hashtable<>();    // Create a new Hashtable with String keys and Integer values

        // Add key-value pairs to the Hashtable using the put() method
        hashtable.put("One", 1);   // Adds "One" -> 1
        hashtable.put("Two", 2);   // Adds "Two" -> 2
        hashtable.put("Three", 3); // Adds "Three" -> 3

        // Retrieve an enumeration of all the keys in the Hashtable using the keys() method
        Enumeration<String> keys = hashtable.keys();

        // Iterate through the keys using a while loop
        while (keys.hasMoreElements()) {
            // Get the next key from the enumeration
            String key = keys.nextElement();    //The first call to nextElement() retrieves the first element in the sequence and moves the internal pointer to the next position. After each call to nextElement(), the internal pointer moves forward, so the next call will return the subsequent element.

            // Use the get() method to retrieve the value associated with the current key
            System.out.println("Key: " + key + ", Value: " + hashtable.get(key));
        }
    }
}
```

Sample Output:

```
Key: One, Value: 1
Key: Two, Value: 2
Key: Three, Value: 3
```

Explanation:

- `Hashtable` stores key-value pairs.
- `Enumeration` is used to iterate through the keys of the `Hashtable`.

The `Enumeration` interface is an older feature from the early days of Java (introduced in JDK 1.0) and provides two primary methods:

- **`hasMoreElements()`** : Checks if there are more elements to iterate over.
- **`nextElement()`** : Retrieves the next element in the enumeration.

Limitations of `Enumeration`:

1. It only supports **forward traversal**, meaning we cannot move backward through the elements.
2. It does **not allow removal** of elements during iteration, unlike the more modern `Iterator` interface.

Modern Alternatives:
In modern Java, the `Iterator` interface is preferred over `Enumeration` because it provides additional functionality, such as the ability to **remove elements** during iteration.

Lab 5: Random Number Generation:

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        Random random = new Random();
        System.out.println("Random Integer: " + random.nextInt(100));
        System.out.println("Random Double: " + random.nextDouble());
        System.out.println("Random Boolean: " + random.nextBoolean());
    }
}
```

Sample Output:

```
Random Integer: 42
Random Double: 0.123456789
Random Boolean: true
```

Explanation:

- `Random` generates random numbers.
- `nextInt()` generates a random integer within a specified range.
- `nextDouble()` generates a random double between 0.0 and 1.0.
- `nextBoolean()` generates a random boolean value.

Summary

- The `java.lang` package provides essential classes like `Math` and wrapper classes for primitive types.
- Wrapper classes allow primitive types to be used as objects and provide utility methods.
- The `java.util` package includes core classes like `Vector`, `Stack`, `Hashtable`, and `Random`.