Unit IV: Generics, Lambdas, GUI Swing & Database Connectivity

Generics Fundamentals, Generic Class, Generic Methods, Lambdas, Functional Interfaces, Swing, Components and Containers, Layout Managers, Swing Event Handling, Event Listeners, Event Classes and Listener Interfaces, Swing Controls, Database Connectivity, Statement, Prepared Statement, CallableStatement, Resultset, Persistent Data.

Functional Interfaces

```java
 * @author dhiviyarj
 */
//Interface with only one abstract method is called Functional Interface
public interface AnonymousInterface {
    void AnonyPrint();

}
```

Key Points About Functional Interfaces

- ✅ A functional interface must have **exactly one abstract method**.
- ✅ Can have **default** and **static** methods.
- ✅ Marked with `@FunctionalInterface` (optional, but helps catch errors).
- ✅ Enables **Lambda Expressions** and **Method References**.

Write a code to implements this functional Interface

```
 * @author dhiviyarj
 */
//Interface with only one abstract method is called Functional Interface
public interface AnonymousInterface {
    void AnonyPrint();

}
```

```
 * @author dhiviyarj
 */
public class AnonyInterfaceMain implements AnonymousInterface{
    @Override
    public void anonyPrint() {
      System.out.println("Anonymous Print");
            }
    public static void main(String[] args) {
      AnonyInterfaceMain obj= new AnonyInterfaceMain();
      obj.anonyPrint();

    }

}
```

If Used only once go for Anonymous class rather than creating new classes

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.

```
 * @author dhiviyarj
 */
public class AnonyInterfaceMain {
    public static void main(String[] args) {
        AnonymousInterface obj= new AnonymousInterface(){
            @Override
            public void anonyPrint()
            {
                System.out.println("Anonymous Print");
            }
        };

       // call the overridden object
       obj.anonyPrint();

    }

}
```

Anonymous Classes

Functional Interface

Anonymous Inner Class

- An class with no name
- Used while extending or implementing if used only in a place

```
 * @author dhiviyarj
 */
//Interface with only one abstract method is called Functional Interface
public interface AnonymousInterface {
    void AnonyPrint();

}
```

```
 * @author dhiviyarj
 */
public class AnonyInterfaceMain {
    public static void main(String[] args) {
        AnonymousInterface obj= new AnonymousInterface(){
            @Override
            public void anonyPrint()
            {
                System.out.println("Anonymous Print");
            }
        };

      // call the overridden object
      obj.anonyPrint();

    }

}
```

Lambda Operator (->) Arrow Token

- Special Operator helps to define a method without the function header

void print(int a,intb) {……function body }

Replaced

(int a,int b) -> { function body }

- They also make the code writing simple, short and readable.
- **In functional interface, instead of using anonymous inner classes and override the method header and body, you can use Lambdas.**

Example for Lambda Operator

```
 * @author dhiviyarj
 */
public class AnonyClassMain {
    public static void main(String[] args)
    {
    //CASE 1: Usage Anonymous Class objects & function call
    AnonymousInterface obj=new AnonymousInterface(){
            @Override
            public void anonyPrint(){
                System.out.println("This is anony print");
            }
        };
    obj.anonyPrint();     // function call

    //CASE 2: Usage of lambda operation
    AnonymousInterface obj2=()->{ System.out.println("Anony Print Using Lambda"); };
    obj2.anonyPrint();

    }

}
```

Being a functional interface it's know it has only one methods so code can be further simplified using lambda

Lambda operation Format

(Arguments) -> {function body}

Functional Interfaces

- General and abstract
- Have only one functional method (default/static)
- Inherited methods are not counter

Lambda with ‘n’ no of arguments and return type

```
 * @author dhiviyarj
 */
public interface TwoArguFuncInterface {
    public void function2(int x,int y);
}
```

```
 * @author dhiviyarj
 */
public class NoOfArguLamdaDemoMain {
    public static void main(String[] args) {
        TwoArguFuncInterface obj= (int x,int y)-> {
            int z;
            z=x+y;
            System.out.println("This is two arguments");
            System.out.println("The value of addition is "+z);
            };
        obj.function2(10, 20);
    }

}
```

```
ut - Run (NoOfArguLamdaDemoMain)
Changes detected - recompiling the module!
Compiling 10 source files to C:\Users\dhiviyarj\Documents\NetBeansProjects\Inner

--- exec-maven-plugin:3.0.0:exec (default-cli) @ InnerClassDemo ---
This is two arguments
The value of addition is 30
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

```
 * @author dhiviyarj
 */
public interface TwoArguFuncInterface {
    public int function2(int x,int y);
}
```

```
 * @author dhiviyarj
 */
public class NoOfArguLamdaDemoMain {
    public static void main(String[] args) {
        TwoArguFuncInterface obj= (int x,int y)-> {
            return(x+y);
            };
        System.out.println(obj.function2(10, 20));
    }
}
```

```
- Run (NoOfArguLamdaDemoMain)
--- exec-maven-plugin:3.0.0:exec (default-cli) @ InnerClassDemo
30
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

- Introduced in JAVA 8
- Reduced Memory Footprints
- Code Optimization
- Functionality passed similar to argument

Need for Generic Programming

Function Overloading - The logic is exactly the same, but the data type is different.

```
int maximum(int a, int b, int c)
 {
      int max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

```
float maximum(float a, float b, float c)
 {
      float max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

Generic Programming in C++

Function Overloading - The logic is exactly the same, but the data type is different.

```
int maximum(int a, int b, int c)
 {
     int max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

```
float maximum(float a, float b, float c)
 {
     float max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

```
Template <class T>
T maximum(T a, T b, T c)
 {
     T max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

Generic Programming in Java

```
public class CacheString {
  private String message;
    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
1 public class CacheAny <T>{
2
3    private T t;
4
5    public void add(T t){
6        this.t = t;
7    }
8
9    public T get(){
10       return this.t;
11   }
12 }
```

```
//Generics
CacheAny<String> myGenericMessage = new CacheAny<String>();
CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();
```

```
public class C
    private Sh

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```

Type Inference Diamond <>

```
//Generics
CacheAny<String> myGenericMessage = new CacheAny<String>();
CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();
```

In Java 7 … still simplified

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

Syntax of a Generic Method

```
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

- <T> is the generic type parameter.
- T[] array means the method can accept an array of any type T.

```
public class GenericMethodExample {

    // Generic method
    public static <T> void printElements(T[] elements) {
        for (T element : elements) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"Java", "Generics", "Method"};

        printElements(intArray);    // Output: 1 2 3
        printElements(strArray);   // Output: Java Generics Method
    }
}
```

Generic Class

- A **Generic Class** in Java is a class that can work with **different types of data** without rewriting code for each data type.
- It provides **type safety**, **reusability**, and **flexibility**.

Syntax:

```
class ClassName<T> {
    // T can be used as a data type throughout the class
}
```

- T is a type parameter.
- can use T just like any data type in the class.

Example: Generic Class

```
//Generic Class
public class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

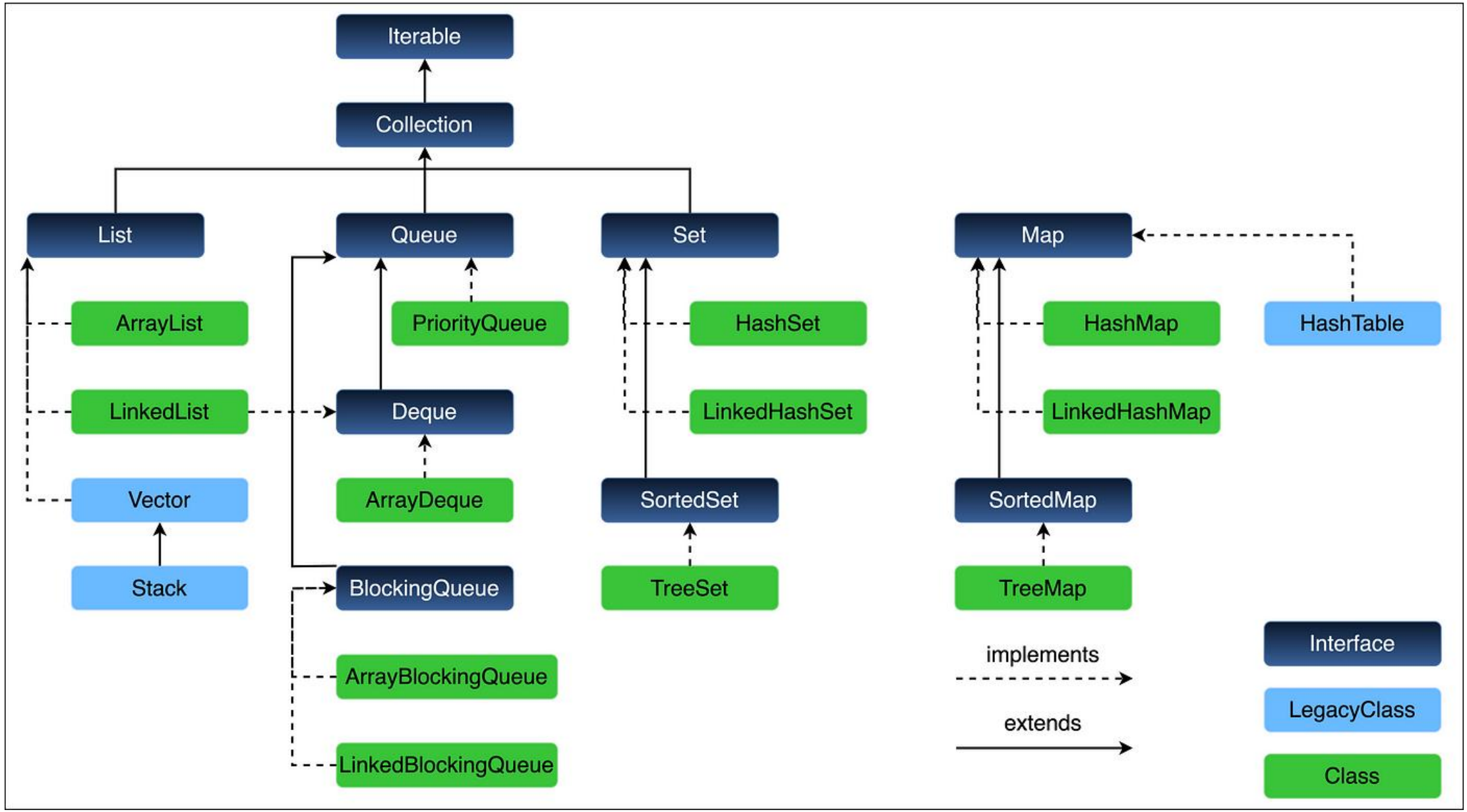
```
//Using Generic Class
public class TestGenericBox {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.set(123);
        System.out.println("Integer Value: " + intBox.get());

        Box<String> strBox = new Box<>();
        strBox.set("Hello");
        System.out.println("String Value: " + strBox.get());
    }
}
```

Generics

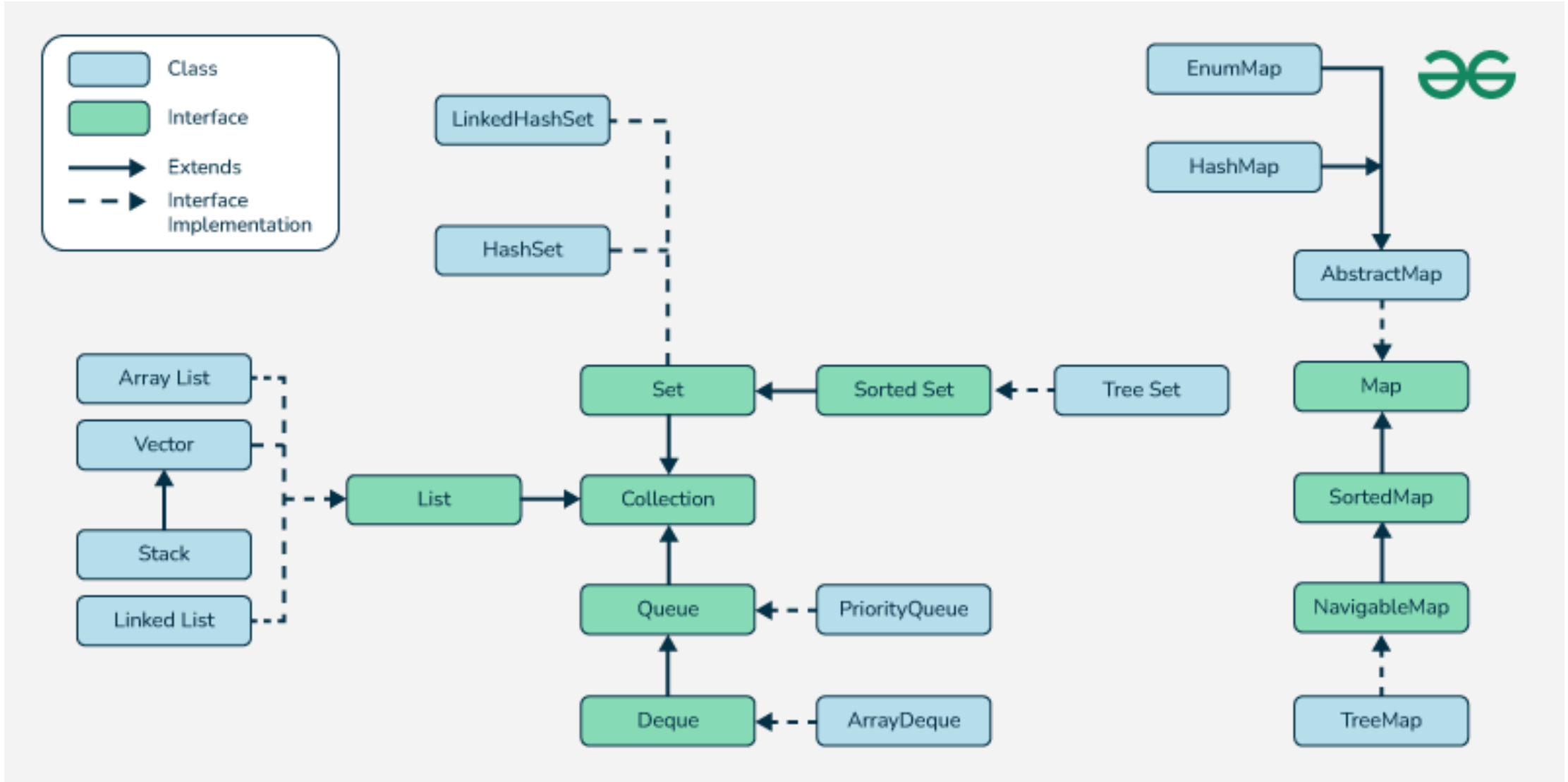
- Type casting are reduced
- Run Time Type Identification
- Generalized Function for Generalized Datatype
- All Java Collections follows Generic Types
- Flexible type safety to code

What are Collections

- A collection is a **holder object that stores a collection of other objects (its elements)**
- It allows a **great flexibility** in the **data types**
- Implements the various **Data Structures**
- Available in **java.util package**


Iterable
Collection
List
Queue
Set
Map
ArrayList
PriorityQueue
HashSet
HashMap
HashTable
LinkedList
Deque
LinkedHashSet
LinkedHashMap
Vector
ArrayDeque
SortedSet
SortedMap
Stack
BlockingQueue
TreeSet
TreeMap
ArrayBlockingQueue
LinkedBlockingQueue
implements
extends
Interface
LegacyClass
Class

Interface	Class Implementations	Features
List	`ArrayList`	Fast for searching, dynamic resizing
	`LinkedList`	Good for insertions/deletions
	`Vector`	Synchronized version of ArrayList
	`Stack`	LIFO (Last In, First Out)
Set	`HashSet`	No duplicates, unordered
	`LinkedHashSet`	No duplicates, maintains insertion order
	`TreeSet`	No duplicates, sorted
Queue	`PriorityQueue`	Elements are ordered by priority
	`ArrayDeque`	Faster than Stack & LinkedList
Map	`HashMap`	Unordered key-value pairs
	`LinkedHashMap`	Maintains insertion order
	`TreeMap`	Sorted key-value pairs
	`Hashtable`	Synchronized version of HashMap


Class
Interface
Extends
Interface Implementation
LinkedHashSet
HashSet
EnumMap
HashMap
AbstractMap
Array List
Vector
Stack
Linked List
List
Set
Sorted Set
Tree Set
Collection
Map
SortedMap
NavigableMap
TreeMap
Queue
PriorityQueue
Deque
ArrayDeque

Feature	List (`java.util.List`)	Set (`java.util.Set`)
Duplicates	Allows duplicates	Does *not* allow duplicates
Order	Maintains insertion order	Does *not* guarantee order (except `LinkedHashSet`)
Access	Can access elements by index (`list.get(index)`)	No index-based access
Null Values	Allows multiple `null` values	Allows only one `null` value
Performance	Faster for searching (`O(1)` with index)	Faster for `contains()` check (`O(1)` in `HashSet`)
Common Implementations	`ArrayList` , `LinkedList`	`HashSet` , `LinkedHashSet` , `TreeSet`

1 Structure of `List` Interface

The `List` interface is part of the **Java Collections Framework** and is present in the `java.util` package. It extends the `Collection<E>` interface.

java

Copy Edit

Explain

```java
public interface List<E> extends Collection<E> {
    // Basic Operations
    boolean add(E e);
    void add(int index, E element);
    boolean remove(Object o);
    E remove(int index);
    E get(int index);
    E set(int index, E element);

    // Search Operations
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
```

List – ArrayList & LinkedList → Code Snippets

UPES
UNIVERSITY OF THE FUTURE

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class ListExample {

 public static void main(String[] args) {
   List<String> al = new ArrayList<String>();
   al.add("Chaitanya");
   al.add("Rahul");
   al.add("Ajeet");
   System.out.println("ArrayList Elements: ");
   System.out.print(al);

   List<String> ll = new LinkedList<String>();
   ll.add("Kevin");
   ll.add("Peter");
   ll.add("Kate");
   System.out.println("\nLinkedList Elements: ");
   System.out.print(ll);
 }
}
```

```
ArrayList Elements:
[Chaitanya, Rahul, Ajeet]
LinkedList Elements:
[Kevin, Peter, Kate]
```

```
List<Integer> partList = new ArrayList<>(3);
    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add(new Integer(4444)); // ArrayList auto grows
    System.out.println("First Part: " + partList.get(0)); //
First item
    partList.add(0, new Integer(5555)); // Insert an item by
index
```

Store a list of integers in an **ArrayList**<Integer> using **autoboxing**. Iterate through the list, unbox each value, and determine if it is a prime number, printing the results.

Create an ArrayList to store Employee objects with attributes like name, id, and salary. Add three employees, update one employee's salary, remove another by their id, and print the remaining employees.

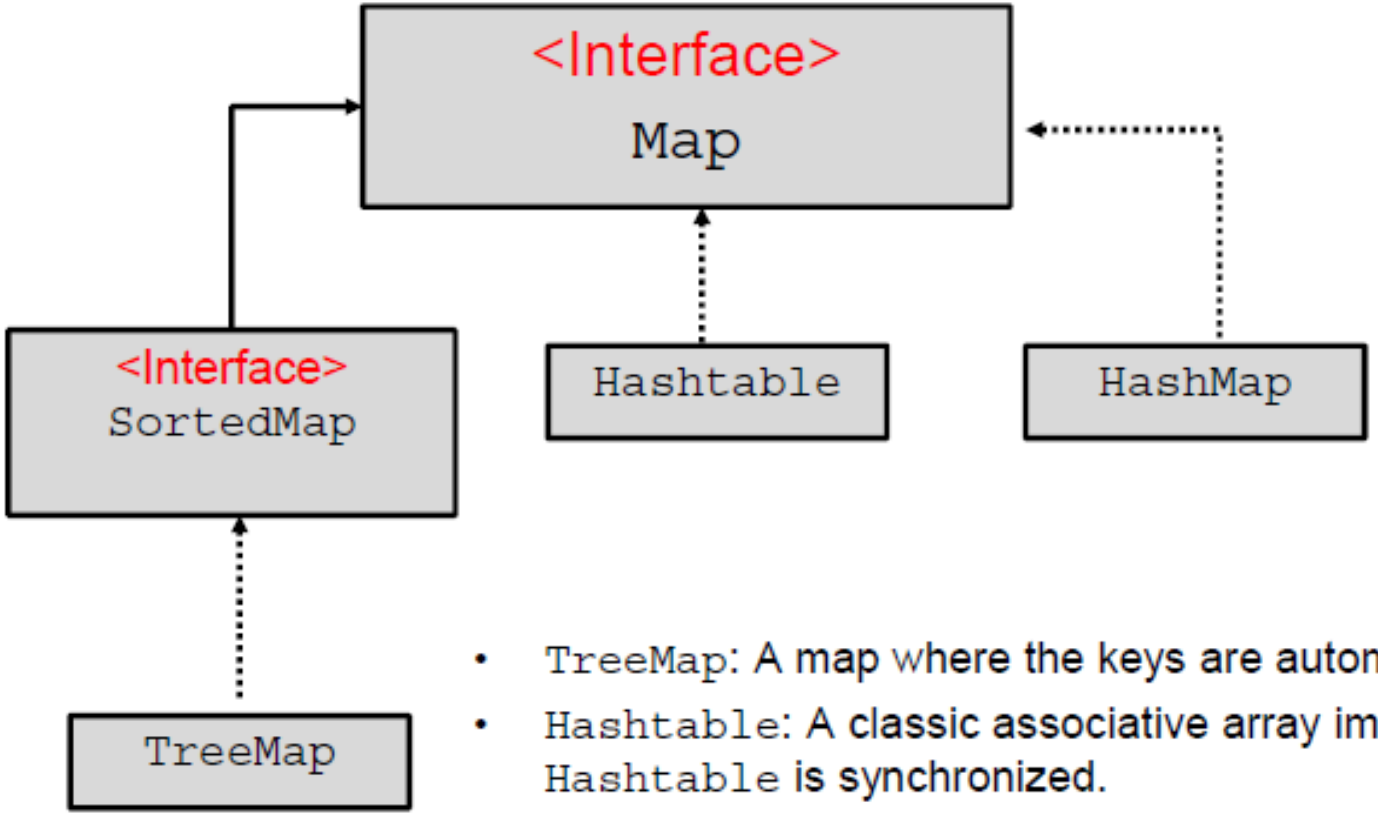
Set - HashSet and TreeSet -> Code Snippet

```java
public class SetExample {

  public static void main(String args[]) {
    int count[] = {11, 22, 33, 44, 55};
    Set<Integer> hset = new HashSet<Integer>();
    try{
      for(int i = 0; i<4; i++){
         hset.add(count[i]);
      }
      System.out.println(hset);

      TreeSet<Integer> treeset = new TreeSet<Integer>(hset);
      System.out.println("The sorted list is:");
      System.out.println(treeset);
    }
    catch(Exception e){
        e.printStackTrace();
    }
  }
}
```

```java
public class SetExample {
   public static void main(String[] args){
        Set<String> set = new TreeSet<>();

        set.add("one");
        set.add("two");
        set.add("three");
        set.add("three"); // not added, only unique

         for (String item:set){
             System.out.println("Item: " + item);
         }
   }
}
```

```
[33, 22, 11, 44]
The sorted list is:
[11, 22, 33, 44]
```

Map are < Key, Value >



- `TreeMap`: A map where the keys are automatically sorted
- `Hashtable`: A classic associative array implementation with keys and values. `Hashtable` is synchronized.
- `HashMap`: An implementation just like `Hashtable` except that it accepts null keys and values. Also, it is not synchronized.

Examples

```
// Set Example
Set<String> sl=new TreeSet<>();
String[] list={"AAA","BBB","CCCC"};
System.out.println("Set Data");
for(i=0;i<3;i++)
sl.add(list[i]);
for(String loop:sl)
{
    System.out.println(loop);
}
```

```
// List Example
System.out.println("The List allows duplication and ordered");
List<Integer> ll=new ArrayList<>();
Integer i;
for(i=10;i<50;i=i+10)
ll.add(i);
System.out.println("List Data");
for(Integer x:ll)
{
    System.out.println(x);
}
```

```
// Map Example
Map<Integer,String> ml=new TreeMap<>();
ml.put(10,"AAA");
ml.put(20,"BBB");
ml.put(30,"CCC");

Set<Integer> index=ml.keySet();
System.out.println(index);

for(Integer x:index)
    System.out.println(x+","+ml.get(x));
```

CUI vs GUI

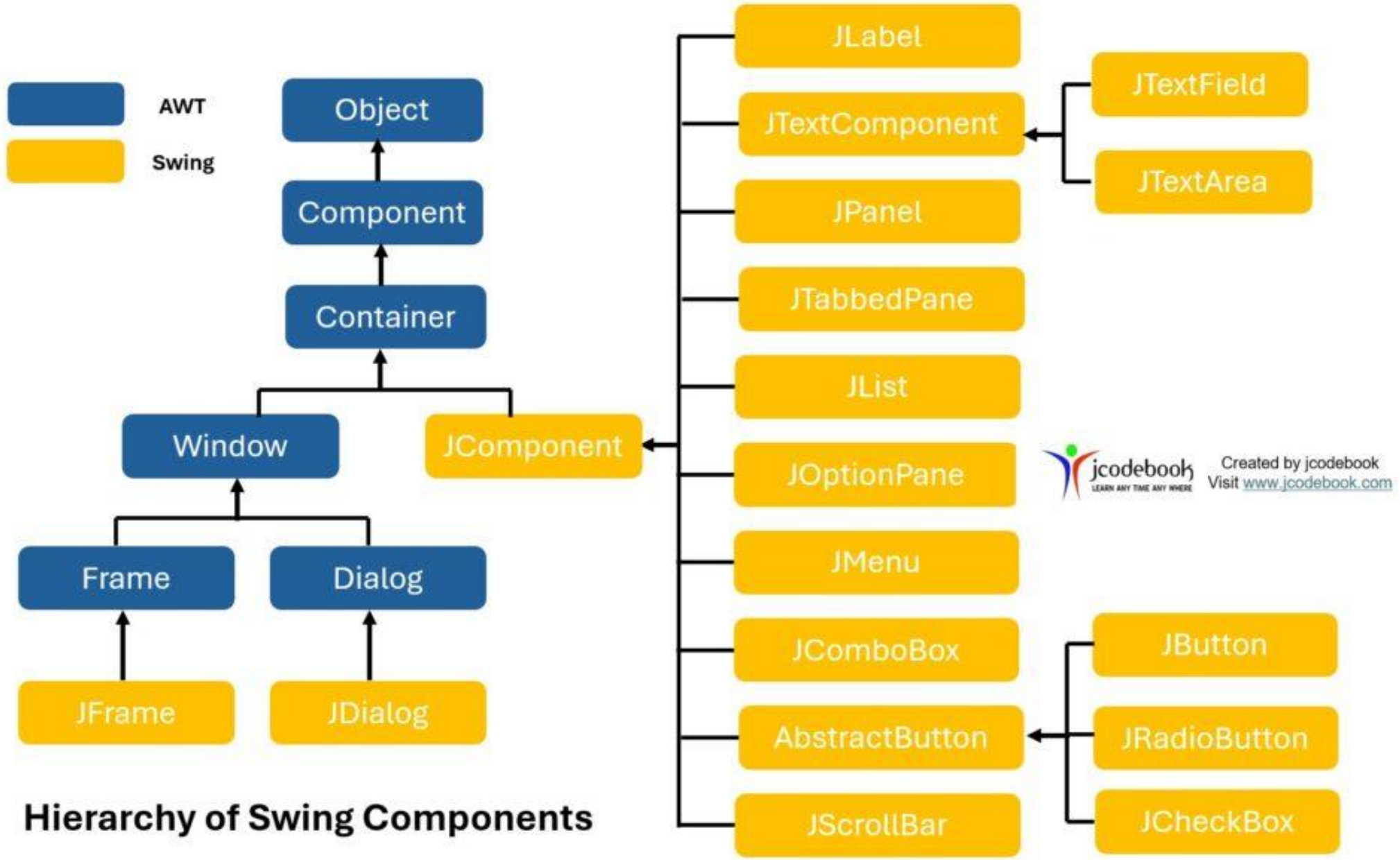
Give the output for a CUI calculator and GUI Calculator

Ways how can I

create a GUI Calculator

Difference between AWT and Swing

Feature	AWT	Swing
Look & Feel	Native	Customizable
Performance	Fast	Moderate
Components	Basic	Rich
Styling	Minimal	Custom via UIManager



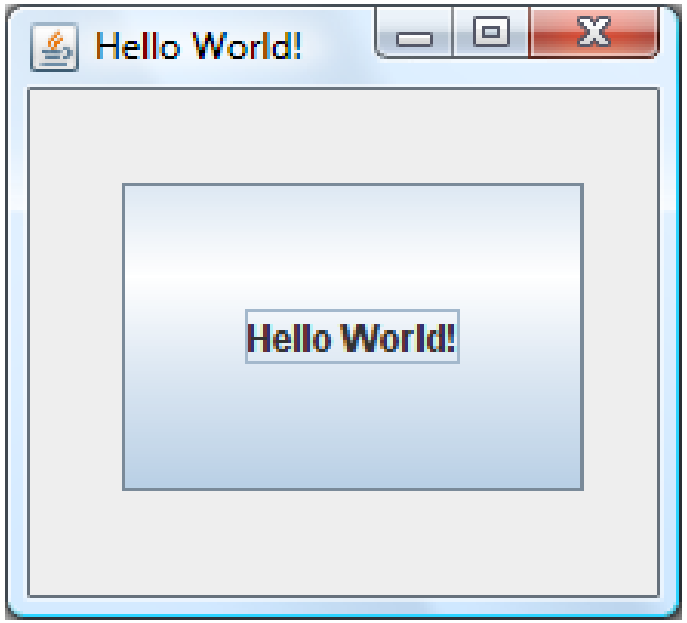
Hierarchy of Swing Components

Swing Design Principles

- GUI is built as **containment hierarchy** of widgets
  (i.e. the parent-child nesting relation between them)
- Event objects and event listeners
  - **Event object**: is created when event occurs (e.g. click), contains additional info (e.g. mouse coordinates)
  - **Event listener**: object implementing an interface with an event handler method that gets an event object as argument
- Separation of Model and View:
  - **Model**: the data that is presented by a widget
  - **View**: the actual presentation on the screen

Swing Hello World

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloWorld {
  public static void main(String[] args) {
        JFrame frame = new JFrame("Hello World!");
        frame.setSize(220, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Container contentPane = frame.getContentPane();
        contentPane.setLayout(null);

        JButton button = new JButton("Hello World!");
        button.setLocation(30, 30);
        button.setSize(150, 100);
        contentPane.add(button);

        frame.setVisible(true);
  }
}
```



✅ Steps to Handle Button Click in Swing

1. **Create a** `JButton`
2. **Attach an** `ActionListener` to the button
3. **Override the** `actionPerformed()` **method** to define what happens when the button is clicked

🎯 Different Ways to Write ActionListener

Separate Class, Anonymous Class, and Lambdas.

1 Using an Anonymous Inner Class (Common Approach)

```
// Add ActionListener to the button
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame, "Button Clicked!");
    }
});
```

2 Using a Separate Class (Best for Multiple Buttons)

```
class ButtonClickHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was clicked!");
    }
}
```

```
button.addActionListener(new ButtonClickHandler());
```

3 Using Lambda Expression (Java 8+) *(Recommended for Simplicity)*

```
button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Button Clicked!"));
```

Events, Listeners, Adapters and Handler Methods

Event	Listener / Adapter	Handler Methods
`ActionEvent`	`ActionListener`	`actionPerformed`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged`

`MouseEvent`	`MouseListener`
	`MouseAdapter`
		`mouseClicked`
		`mouseEntered`
		`mouseExited`
		`mousePressed`
		`mouseReleased`

`KeyEvent`	`KeyListener`
	`KeyAdapter`
		`keyPressed`
		`keyReleased`
		`keyTyped`

`ComponentEvent`	`ComponentListener`
	`ComponentAdapter`
		`componentShown`
		`componentHidden`
		`componentMoved`
		`componentResized`

Adapter classes with empty methods for Listener interfaces with >1 methods

Write a Java Swing program to create a Login Form using JTextField, JPasswordField, JButton, and JLabel. When the login button is clicked, validate if the username is "admin" and the password is "password".

Design a simple calculator using Java Swing with buttons for digits (0-9), addition (+), subtraction (-), multiplication (*), and division (/). Implement event handling for button clicks and display the result in a JTextField.

Create a To-Do List Application

Write a Java Swing program to implement a To-Do List using JList.

Provide:

- ✓ A JTextField to enter tasks
- ✓ An Add button to add tasks to the list
- ✓ A Remove button to delete selected tasks

Thank You