# 11. Collections & Generics

## 11.1) Variable arguments

- Java's varargs feature allows methods to accept any number of arguments.
- It is declared using an ellipsis (...).
- Internally, varargs are treated like arrays.
- A method using varargs can still be called even if no arguments are passed.
- If you want to require at least two arguments, you can declare two fixed parameters followed by varargs:

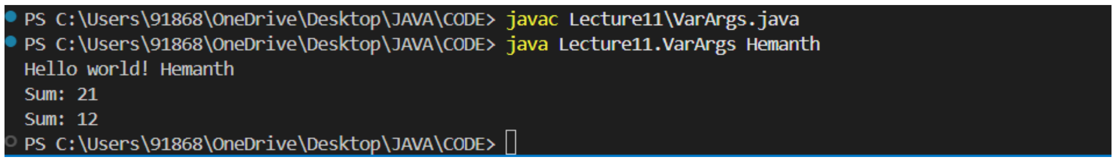
```java
void sum(int a, int b, int... nums){


}
```

- Here, a and b are mandatory, and nums can accept any additional arguments.
- **Varargs must always be the last parameter in the method signature.**
- **Its also valid to write String... args as the main method signature.**

```java
package Lecture11;

public class VarArgs {
    public static void main(String... args) {
        System.out.println("Hello world! " + args[0]);
        VarArgs obj = new VarArgs();
        obj.sum(1, 2, 3, 4, 5, 6);
        obj.sum(5, 7);
    }
    public void sum(int... nums) {
        int sum = 0;
        for (int var : nums) {
            sum += var;
        }
        System.out.println("Sum: " + sum);
    }
}
// Hello world!
// Sum: 21
// Sum: 12
```



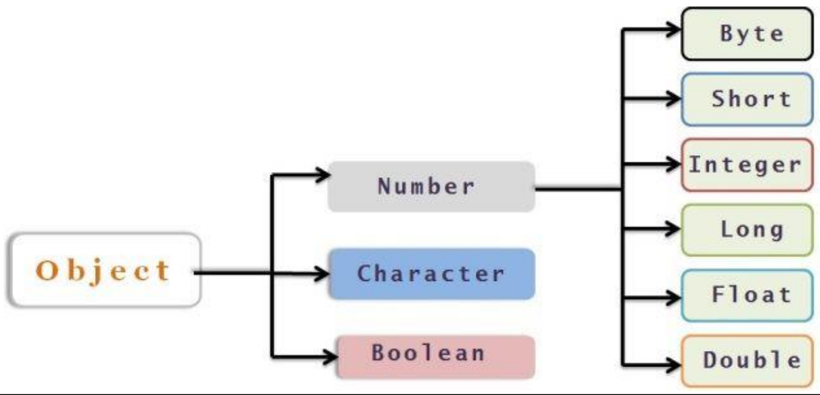
```
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE> javac Lecture11\VarArgs.java
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE> java Lecture11.VarArgs Hemanth
Hello world! Hemanth
Sum: 21
Sum: 12
PS C:\Users\91868\OneDrive\Desktop\JAVA\CODE>
```

## 11.2) Wrapper classes

- Using wrapper classes we can use primitive datatypes as objects.
- Automatic conversion between the primitive types and their corresponding wrapper classes.
- Once created, value of a wrapper object can not be changed.
- Provides useful methods like valueOf(), compareTo(),...
- Mainly used to store primitives in collection objects like ArrayList, HashMap,....
- We can assign null to primitive values if needed.

```
int a = null; // invalid
Integer b = null; // valid
```

- AutoBoxing is the automatic conversion of  primitive types to their corresponding wrapper class objects.
- Unboxing is the automatic conversion of wrapper class objects back to their respective primitive types.



```
package Lecture11;

public class WrapperClass {
    public static void main(String[] args) {
        Integer num = Integer.valueOf("55"); // Converts a String to an Integer object
        Integer num2 = 54; // Autoboxing: primitive int to Integer
        Integer num3 = Integer.valueOf(12); // Explicit boxing
        int primitiveNum = num2; // unboxing: Integer to int
        int sum = primitiveNum + num3;
        // Output
        System.out.println("num  = " + num);
        System.out.println("num2 = " + num2);
        System.out.println("num3 = " + num3);
        System.out.println("primitiveNum (unboxed from num2) = " + primitiveNum);
        System.out.println("Sum of primitiveNum and num3 = " + sum);
    }
}
// num = 55
// num2 = 54
// num3 = 12
// primitiveNum (unboxed from num2) = 54
// Sum of primitiveNum and num3 = 66
```

## 11.3) Collections library

- Collections are groups of similar elements.
- They support dynamic memory allocation, unlike arrays which are fixed in size.
- The Collection and Map interfaces are part of the Java Collections Framework.
- The Collection interface is the root interface of the collection hierarchy and provides basic operations like: add(), remove(), clear(), and size().
- Any class that overrides these methods can be considered a collection type.

- We can create custom collections by implementing the Collection interface (or a more specific interface such as List, Set, or Queue) in our class and overriding methods such as add(), remove(), clear(), and size().
- A LinkedList in Java is an example of multiple inheritance via interfaces, as it implements both the List and Queue interfaces.

- List interface is an ordered collection that can contain duplicate elements.
- Set interface is a collection that cannot contain duplicate elements.
- Queue interface  is a collection used to hold elements in FIFO.
- Map interface is not truly a collection, but part of the collections framework and stores the key-value pairs where keys are unique but not the values.

## 11.4) List Interface

- List is an ordered collection that allows duplicates.
- Elements can be accessed by their integer index and also maintains the insertion order of elements.
- Grows automatically as elements are added, offers fast random access and quick iteration.
- Preferred over arrays as size is dynamic.

```java
import java.util.ArrayList;

public class ListDemo {
    public static void main(String[] args) {
        // Create a new ArrayList to store Integer values
        ArrayList<Integer> list = new ArrayList<>();
        // Adding elements to the list
        list.add(10); // list: [10]
        list.add(30); // list: [10, 30]
        list.add(40); // list: [10, 30, 40]
        list.add(1, 20); // Inserts 20 at index 1 -> list: [10, 20, 30, 40]
        printList(list); // Output: 10, 20, 30, 40

        // Removing elements
        list.remove(0); // Removes element at index 0 (10) -> list: [20, 30, 40]
        list.remove(Integer.valueOf(40)); // Removes the value 40 -> list: [20, 30]

        printList(list); // Output: 20, 30

        // Accessing and modifying elements
        System.out.println(list.get(0)); // Gets the element at index 0 -> Output: 20
        list.set(0, 30); // Replaces element at index 0 with 30 -> list: [30, 30]
        // Note: set() cannot add a new element; it only modifies an existing one

        printList(list); // Output: 30, 30

        // Checking for presence and position of an element
        System.out.println(list.contains(25)); // false (25 not in list)
        System.out.println(list.indexOf(25)); // -1 (25 not found)

        list.clear(); // Removes all elements -> list: []

        printList(list); // Output: empty
    }
    public static void printList(ArrayList<Integer> list) {
        // for (int i : list) {
        // System.out.println(i);
        // }
        // OR
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        System.out.println("-----------------------");
    }
}
```

```
// 10
// 20
// 30
// 40
// -----------------------
// 20
// 30
// -----------------------
// 20
// 30
// 30
// -----------------------
// false
// -1
// -----------------------
```

- <> are used in Java generics to help developers catch type-related errors at compile time. These are known as diamond operators or diamond brackets.
- After compilation, the compiler performs a process called type erasure, which removes the generic type information. This means the type parameters (inside <>) do not exist at runtime.

**Generics are used at compile time only** to enforce type safety. Example:

```
List li = new ArrayList();
```

The above line is valid in Java, but it creates a raw type. This means the list can store any type of object (e.g., Integer, String, etc.), but it disables compile-time type checking and may lead to runtime ClassCastException. It's recommended to use generics like this:

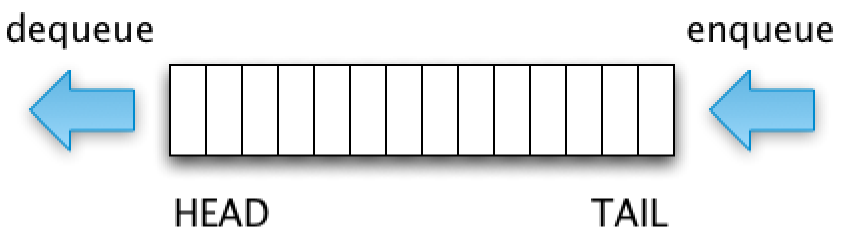
```
List<String> li = new ArrayList<>();
```

This ensures that only String objects can be added to the list, providing better type safety and avoiding casting issues.

| **Method** | **Description** |
|---|---|
| add(E e) | ⟶ Adds an element to the list at the end. |
| add(int index, E element) | ⟶ inserts an element at the specified index, shifting elements to the right. |
| remove(int index) | ⟶ Removes the element at the specified index. |
| remove(Object o) | ⟶ Removes the first occurrence of the specified element. |
| get(int index) | ⟶ Retrieves the element at the specified index. |
| set(int index, E element) | ⟶ Replaces the element at the specified index with the given element. |
| contains(Object o) | ⟶ Checks if the list contains the specified element. |
| indexOf(Object o) | ⟶ Returns the index of the first occurrence of the specified element. |
| clear() | ⟶ Removes all elements from the list. |
| size() | ⟶ Returns the number of elements currently in the list. |

## 11.5) Queue Interface

- Folllows FIFO.
- There are 2 ends - one for insertion and the other for removal.



| **Method** | **Description** |
|---|---|
| add(E e)<br>offer(E e) | ⟶ inserts specific element, throws exception if it can not be added.<br>⟶ Same as above but returns false if the element can not be added. |
| remove()<br>poll() | ⟶ retreives and removes the head of the queue, throws exception if queue is empty.<br>⟶ same as above but returns null if queue is empty. |
| element()<br>peek() | ⟶ Retrieves but doesnot remove the head of the queue, throws exception if queue is empty.<br>⟶ Same as above but returns null if queue is empty. |

Since we need to print collections multiple times, we define a custom utility class named Hemanth with a static method print(). This method accepts a Collection object as an argument, which allows it to handle any type of collection (like List, Queue, Set, etc.) because Collection is a common parent interface.

```java
import java.util.Collection;

public class Hemanth {
    public static void print(Collection obj) { // since Collection is a parent class, it can accept any of its child
        System.out.print("Collection is: "); // class as parameter.
        for (Object a : obj) {
            System.out.print(a + " ");
        }
        System.out.println();
    }
}
```

```java
import java.util.LinkedList;
import java.util.Queue;

public class TestingQueue {
    public static void main(String[] args) {
        Queue<Integer> q = new LinkedList<>();
        q.add(10);
        q.offer(20);
        Hemanth.print(q);
        System.out.println(q.peek());
        System.out.println(q.element());
        Hemanth.print(q);
        System.out.println(q.poll());
        Hemanth.print(q);
        System.out.println(q.remove());
        Hemanth.print(q);
        System.out.println(q.poll());
    }
}
```

```
// Collection is: 10 20
// 10
// 10
// Collection is: 10 20
// 10
// Collection is: 20
// 20
// Collection is:
// null
```

## 11.6) Set Interface

- Contains unique elements.
- Doesnot guarentees any specific order of elements.
- No positional access I.e doesnot support indexing.
- Common implementations are HashSet, LinkedHashSet, and TreeSet(maintains order and uniqueness).
- The contains() method in a Set is usually faster than in a List.

| Method | Description |
|---|---|
| add(E e) | ⟶ Returns true if the element was added, false if it already exists |
| remove(Object o) | ⟶ Returns true if the element was present and removed, else false. |
| contains(Object o) | ⟶ Returns true if the element exists in the set. |
| size() | ⟶ returns no.of elements in the set. |
| isEmpty() | ⟶ Returns true if the set has no elements, else false. |

```java
import java.util.HashSet;
import java.util.Set;

public class TestingSet {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("Hemanth"));
        System.out.println(set.add("Line2"));
        System.out.println(set.add("Apple"));
        System.out.println(set.add("Apple"));
        Hemanth.print(set);
        System.out.println(set.size());
    }
}
```

```
// true
// true
// true
// false
// Collection is: Apple Hemanth Line2
// 3
```

## 11.7) Collections class

Collections is an utility class where Collection is an interface.

Offers different predefined static methods to perform operations like sort, max, min ,reverse, binary searching, shuffling ….

We can also make an unmodifiable list.

```java
List<Integer> finalList = Collections.unmodifiableList(list);
        finalList.add(10);
        Hemanth.print(finalList);
        // java.lang.UnsupportedOperationException
```

Use of some of the methods are

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

public class CollectionsTesting {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(5);
        list.add(-7);
        Hemanth.print(list);
        Collections.sort(list);
        Hemanth.print(list);
        System.out.println(Collections.max(list));
        System.out.println(Collections.min(list));
        Collections.reverse(list);
        Hemanth.print(list);
        System.out.println(Collections.binarySearch(list, 5));
        List<Integer> permanentList = Collections.unmodifiableList(list);// read-only list
    }
}
```

```
// Collection is: 10 5 -7
// Collection is: -7 5 10
// 10
// -7
// Collection is: 10 5 -7
// 1
```

## CHALLENGES

1) Write a method concatinate strings that takes variable arguments of string type and concatinates them into a single string.

```java
public class Challenge1 {
    public static void concatinateString(String... lines) {
        StringBuilder sb = new StringBuilder();
        for (String line : lines)
            sb.append(line).append(" ");
        System.out.println("Final String: " + sb.toString());
    }

    public static void main(String[] args) {
        concatinateString("Hello", "My", "Name", "is", "Hemanth");
    }
}
```

```
// Final String: Hello My Name is Hemanth
```

2) Write a program that sorts a list of String objects in descending order using a custom comparator.

```java
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Arrays;;

public class ComparatorDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("C", "B", "A", "Z");
        Hemanth.print(list);
        sortDescending(list);
    }
    public static void sortDescending(List<String> list) {
        Collections.sort(list, new Comparator<String>() {
            @Override
            public int compare(String str1, String str2) {
                if (str1.equals(str2))
                    return 0;
                else if (str1.charAt(0) < str2.charAt(0))
```

```
                    return 1;
                else
                    return -1;
            }
        });
        Hemanth.print(list);
    }
}
```

```
Collection is: C B A Z
Collection is: Z C B A
```

OR

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

public class ListOfStrings {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("C");
        list.add("A");
        list.add("B");
        list.add("D");
        Hemanth.print(list);
        Collections.sort(list);
        Hemanth.print(list);
        Collections.reverse(list);
        Hemanth.print(list);
    }
}
```

```
// Collection is: C A B D
// Collection is: A B C D
// Collection is: D C B A
```

3) Use the collections class to count the frequency of a particular element in an arraylist.

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

public class FrequencyCount {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(2);
        Hemanth.print(list);
        System.out.println("Frequency of 2: " + Collections.frequency(list, 2));
    }
}
```

```
// Collection is: 1 2 3 2
// Frequency of 2: 2
```

4) Write a method to swap 2 elements in an ArrayList, given their indices.

```
import java.util.ArrayList;
```

```java
import java.util.List;

public class Swap {
    public static void swap(List<String> list, int indx1, int indx2) {
        String temp1 = list.get(indx1);
        String temp2 = list.get(indx2);
        list.set(indx1, temp2);
        list.set(indx2, temp1);
        System.out.print("After ");
        Hemanth.print(list);
    }
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("one");
        list.add("Two");
        list.add("Three");
        System.out.print("Before ");
        Hemanth.print(list);
        swap(list, 0, 2);
    }
}
```

```
// Before Collection is: one Two Three
// After Collection is: Three Two one
```

5) Create a program that reverses the elements of a list and prints the reversed list.

```java
import java.util.List;
import java.util.Arrays;

public class Reverse {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
        for (int i = 0, j = list.size() - 1; i < j; i++, j--) {
            int temp1 = list.get(i);
            int temp2 = list.get(j);
            list.set(j, temp1);
            list.set(i, temp2);
        }
        System.out.println(list); // [6, 5, 4, 3, 2, 1]
    }
}
```

OR

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public class ReversedList {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3);
        System.out.print("Before ");
        System.out.println(list);
        Collections.reverse(list);
        System.out.print("After ");
        System.out.println(list);
    }
}
```

```
// Before [1, 2, 3]
// After [3, 2, 1]
```

6) create a PriorityQueue of a custom class Student with attributes name and grade. Use a comparator to order by grade.

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class Student1 {
    public static class InnerStudent1 {
        private final String name;
        private final char grade;
        public InnerStudent1(String name, char grade) {
            this.name = name;
            this.grade = grade;
        }
        public String getName() {
            return name;
        }
        public char getGrade() {
            return grade;
        }
        @Override
        public String toString() {
            return name + " : " + getGrade();
        }
    }
    public static void main(String[] args) {
        PriorityQueue<InnerStudent1> q = new PriorityQueue<>(new Comparator<InnerStudent1>() {
            @Override
            public int compare(Student1.InnerStudent1 o1, Student1.InnerStudent1 o2) {
                return o1.getGrade() - o2.getGrade();
            }
        });
        q.offer(new InnerStudent1("Hemanth", 'E'));
        q.offer(new InnerStudent1("Amrish", 'C'));
        q.offer(new InnerStudent1("Nobitha", 'F'));
        q.offer(new InnerStudent1("Deskisuki", 'A'));
        System.out.println(q);
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
        System.out.println(q.poll());
    }
}
// [Deskisuki : A, Amrish : C, Nobitha : F, Hemanth : E]
// Deskisuki : A
// Amrish : C
// Hemanth : E
// Nobitha : F
// null
```

7) Write a program that takes a string and returns the number of unique characters using set.

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
```

```java
public class Unique {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String name = s.next();
        Set<Character> set = new HashSet<>();
        for (int i = 0; i < name.length(); i++)
            set.add(name.charAt(i));
        System.out.println("Number of unique elements are: " + set.size());
    }
}
// Enter a string: Hemanth
// Number of unique elements are: 7
```

OR
we can use the method toCharArray.

```java
char[] charArray = str.toCharArray();
```

## 11.8) Map Interface

Map interface is a part of Collections library/ Framework but not related to Collection interface.

Stores data as key-value pairs.

Each key can map to atmost one value.

Keys are unique, but multiple keys can map to same value.

| Method | Description |
|---|---|
| put(K key, V value) | ⟶ Associates the specified value with the specified key in the map. |
| get(Object key) | ⟶ Returns the value to which the specified key is mapped, or null if no mapping exists. |
| remove(Object key) | ⟶ Removes the mapping for a key if it is present. |
| containsKey(Object key) | ⟶ Checks if the map contains a mapping for the specified key. |
| keySet() | ⟶ Returns a Set view of the keys contained in the map. |
| values() | ⟶ Returns a Collection view of the values contained in the map. |

```java
import java.util.HashMap;
import java.util.Map;

public class MapDemo {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("A", 65);
        map.put("B", 66);
        map.put("C", 67);
        map.put("D", 68);
        System.out.println(map.size());
        System.out.println(map.get("D"));
        System.out.println(map.get("E"));
        System.out.println(map.remove("D"));
        System.out.println(map);
        System.out.println(map.containsKey("D"));
        System.out.println(map.keySet());
        System.out.println(map.values());
        for (String key : map.keySet())
            System.out.printf("%s:%d ", key, map.get(key));
    }
}
```

```
// 4
// 68
// null
// 68
// {A=65, B=66, C=67}
// false
// [A, B, C]
// [65, 66, 67]
// A:65 B:66 C:67
```

## 11.9) Enums

- Enums (short for enumerations) are special types of classes that represent a fixed set of constants.
- They are used when a variable can only take one out of a small set of predefined values (e.g., directions, colors, traffic lights).
- Enum constants are usually written in uppercase letters by convention.
- Enums are defined using the enum keyword.

```
enum Color {
    RED, GREEN, BLUE;
}
```

Semicolon is optional if you are no longer declaring a constructor or an attribute or any  methods.

You can access enum constants using the dot (.) operator:

```
Color myColor = Color.RED;
```

values() ⟶ Returns an array of all enum constants in the order they're declared.

valueOf(String name) ⟶ Returns the enum constant with the specified name (case-sensitive). Throws IllegalArgumentException if the name doesn't match.

```
public enum TreafficLights {
    RED("Stop"), GREEN("Go"), YELLOW("Get ready to go");

    private String action;
    private TreafficLights(String action) {
        this.action = action;
    }
    public String getAction() {
        return action;
    }
}
```

```
import javax.swing.Action;

public class EnumDemo {
    public static void main(String[] args) {
        TreafficLights currentLight = TreafficLights.GREEN;
        System.out.println("current light: " + currentLight);
        System.out.println("Action: " + currentLight.getAction());
        TreafficLights red = TreafficLights.valueOf("RED");
        System.out.println("Light from valueOf(): " + red);
        System.out.println("Action: " + red.getAction());
        System.out.println("All traffic lights and actions:");
        for (TreafficLights light : TreafficLights.values())
            System.out.println(light + " -> " + light.getAction());
    }
}
```

```
// current light: GREEN
// Action: Go
// Light from valueOf(): RED
// Action: Stop
// All traffic lights and actions:
// RED -> Stop
// GREEN -> Go
// YELLOW -> Get ready to go
```

## 11.10) Generics & Diamond Operators

- Generics enable you to write flexible, type-safe, and reusable code.
- They allow types (classes or interfaces) to be parameters when defining classes, interfaces, and methods.
- Generics provide compile-time type checking, reducing the risk of ClassCastException.

```
ArrayList list = new ArrayList();
list.add("hello");
list.add(123);  // No error at compile time

String str = (String) list.get(1);  // Runtime error: ClassCastException
```

```
ArrayList<String> list = new ArrayList<>();
list.add("hello");
list.add(123);  // ❌Compile-time error: incompatible types
```

- With generics, explicit casting is not required, as the type is already known to the compiler.

```
ArrayList list = new ArrayList();
list.add("Java");

String language = (String) list.get(0);  // You must cast manually
```

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");

String language = list.get(0);  // No casting required – clean and safe
```

- Generics are denoted using angle brackets (<>), where you specify the type parameter (e.g., List<String> means a list that holds String values).
- The "diamond operator" (<>) specifically refers to a feature introduced in Java 7 that allows you to omit the type on the right-hand side of a declaration when it can be inferred by the compiler.

Below is the SpecificClass implementation where we can only store integers (or the defined type). To store other types, you'd need separate classes.

```
public class SpecificClass {
    private final int var;

    public SpecificClass(int var) {
        this.var = var;
    }
    public int getVar() {
        return var;
    }
}
```

Below is the generic lass where we can create any type of object , no type cast needed:The type is preserved at compile time (but erased at runtime — known as type erasure).

```
public class GenericClass<T> {
    private final T var;

    public T getVar() {
        return var;
    }
    public GenericClass(T var) {
        this.var = var;
    }
}
```

```

GenericClass<String> stringObj = new GenericClass<>("Hello");
System.out.println(stringObj.getVar()); // Output: Hello

GenericClass<Integer> intObj = new GenericClass<>(42);
System.out.println(intObj.getVar()); // Output: 42
```

**Type Erasure is a process by which Java compiler removes all generic type information during compilation. This means that generic types exist only at compile time, and the compiled bytecode contains non-generic code (i.e., raw types).**

## CHALLENGES

1) Create an enum called Day that represents the days of the week. Write a program that prints out all the days of the week from this enum.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

```
public class DayDemo {
    public static void main(String[] args) {
        for (Day name_of_the_day : Day.values())
            System.out.println(name_of_the_day);
    }
}
```

```
// MONDAY
// TUESDAY
// WEDNESDAY
// THURSDAY
// FRIDAY
// SATURDAY
// SUNDAY
```

2) Enhance the Day enum by adding an attribute that indicates whether it is a weekday or weekend. Add a method in the enum that returns whether it's a weekday or weekend, and write a program to print out each day along with its type.

```
public enum Day {
    MONDAY("Weekday"), TUESDAY("Weekday"), WEDNESDAY("Weekday"), THURSDAY("Weekday"),
        FRIDAY("Weekday"),SATURDAY("Weekend"), SUNDAY("Weekend");

    private final String type;
    private Day(String type) {
        this.type = type;
    }
    public String getType() {
        return type;
    }
}
```

```
public class DayDemo {
    public static void main(String[] args) {
        for (Day name_of_the_day : Day.values())
            System.out.println(name_of_the_day + "->" + name_of_the_day.getType());
    }
}
```

```
// MONDAY->Weekday
// TUESDAY->Weekday
// WEDNESDAY->Weekday
// THURSDAY->Weekday
// FRIDAY->Weekday
// SATURDAY->Weekend
// SUNDAY->Weekend
```

3) Create a map where the keys are country names(as String) and the values are their capitals(also String). Populate the map with at least 5 countries and their capitals. Write a program that prompts user to enter a country name and then displays the corresponding capital, if it exists in the map.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class CountryMap {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("India", "Delhi");
        map.put("Japan", "Tokyo");
        map.put("America", "Washington DC");
        map.put("China", "Beijing");
        map.put("Australia", "Canberra");
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the name of the country: ");
        String prompt = s.nextLine();
        System.out.println("Capital is: " + map.get(prompt));
        if (map.containsKey(prompt))
```

```
            System.out.println("Capital is: " + map.get(prompt));
        else
            System.out.println("can't found in our DB");
    }
}
```

```
// Enter the name of the country: australia
// Capital is: null
// can't found in our DB
```

## KEYPOINTS

- Autoboxing automatically converts the primitive type to its respective wrapper class object. For example, int to Integer, double to Double, etc.
- Not every class in the collections library implements the Collection interface, like Map. Map is part of the Java Collections Framework, but it does not implement the Collection interface.
- List maintains insertion order and allows adding elements at specified positions. Implementations like ArrayList and LinkedList maintain order and allow indexed insertion.
- In Map, keys are unique, but values can be duplicated. A key maps to exactly one value, but multiple keys can map to the same value.
- Enums in Java can implement interfaces and can also have attributes, methods, and constructors. Enums are powerful and can behave much like regular classes.
- With the diamond operator (<>), we don't need to specify the type on the right-hand side when initializing an object. This was introduced in Java 7 for cleaner and type-safe code:

```
List<String> list = new ArrayList<>();
```

- Wrapper classes in Java like Integer and Double cannot be extended because they are final classes. You cannot inherit from these classes.

.