I. OBJECT ASSOCIATION, COMPOSITION AND AGGREGATION

I.1. Association

Association is relation between two separate classes which is established through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

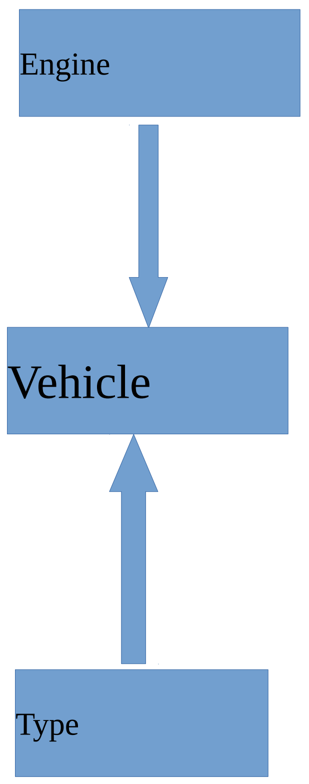
I.2. Composition

Objects Composition is the design technique used in order to implement the has-a relationship in classes in exactly and different way as we can do the same using inheritance. Bot composition and inheritance in java are to enable code reuse.

To achieve composition in java we use instance variables that refers to other objects.

I.2.1 Examples:

1. The relationship between a vehicle ,engine,and Type



2. The relationship Between the Person and The Job he has

3. But not the he relationship between the Victim and the Person

(This is an inheritance)

I.2.2. Sample Codes:(Vehicle , VehicleT ype and Engine)

1. **The Engine Class :**

```
public class Engine {

    private Long id;

    private String version;

    private String name;

    private Long size;

    private Double power;

    public Engine() {

    }

    public Engine(Long id, String version, String name, Long size,
Double power) {

        this.id = id;

        this.version = version;

        this.name = name;

        this.size = size;

        this.power = power;

    }


    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
	public Long getSize() {
		return size;
	}

	public void setSize(Long size) {
		this.size = size;
	}

	public Double getPower() {
		return power;
	}

	public void setPower(Double power) {
		this.power = power;
	}

}
```

2. **Vehicle Type**

```
public class VehicleType {

	private Long id;

	private String name;

	private Long numberOfSeats;


	public VehicleType() {

	}


	public VehicleType(Long id, String name) {


		this.id = id;

		this.name = name;

	}


	public Long getId() {

		return id;

	}


	public void setId(Long id) {

		this.id = id;

	}
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public Long getNumberOfSeats() {
		return numberOfSeats;
	}

	public void setNumberOfSeats(Long numberOfSeats) {
		this.numberOfSeats = numberOfSeats;
	}

}
```

3. **The Vehicle Class**

```
public class Vehicle {
```

```
private Long id;

private String numberPlate;


private VehicleType type;


private Engine engine;


public Vehicle() {}


public Vehicle(Long id,String numberPlate,VehicleType type) {

        this.id=id;

        this.numberPlate=numberPlate;

        this.type=type;

}


public Vehicle(Long id,String numberPlate,VehicleType type,Engine
engine) {

        this.id=id;

        this.numberPlate=numberPlate;

        this.type=type;

        this.engine=engine;

}
```

```
public Long getId() {

        return id;

}


public void setId(Long id) {

        this.id = id;

}


public String getNumberPlate() {

        return numberPlate;

}


public void setNumberPlate(String numberPlate) {

        this.numberPlate = numberPlate;

}


public VehicleType getType() {

        return type;

}


public void setType(VehicleType type) {

        this.type = type;
}

public Engine getEngine() {
	return engine;
}

public void setEngine(Engine engine) {
	this.engine = engine;
}



}
```

I.2.3 Characteristics of compositions

It is a special form of Association where:

- It represents **Has-A** relationship.
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.

- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity

I.3. Aggregation vs Composition

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
3. **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Note: Composition and Aggregation will be explained when done with collections

II. ABSTRACT CLASSES & INTERFACES

II.1. What is an Abstract class ?

An **Abstract class** in Java ,is a special class marked buy the **abstract** keyword and composed by abstract methods **ie**. methods without body and methods with implementation.

The **abstract** keyword is used to create an abstract class and method. Abstract class in java can’t be instantiated. An abstract class is mostly used to provide a base for sub classes to extend and implement the abstract methods and override or use the implemented methods in abstract class.

II.1. 1. Example of an Abstract Class:

```
public abstract class Person {

	private String name;
	private String gender;
	public Person(String nm, String gen) {
		this.name = nm;
		this.gender = gen;
	}

	// abstract method
```

```
        public abstract void work();

        @Override
        public String toString() {
                return "Name=" + this.name + "::Gender=" + this.gender;
        }

        public void changeName(String newName) {
                this.name = newName;
        }
}
```

Notice that the **work()** method is an abstract and it **has no-body**.

II.1.2 Implementing Abstract class example:

```
public class Employee extends Person {

     private String ettendanceNumber;

     public Employee(String name, String gender, String ettendanceNumber) {
               super(name, gender);
               this.ettendanceNumber=ettendanceNumber;
     }

     @Override
     public void work() {
               if(ettendanceNumber == null){
                   System.out.println("Not yet started to work");
               }else{
                   System.out.println("Allready working!");
               }
     }

     public static void main(String args[]){
               //using  abstract classes
               Person student = new Employee("Mugisha","Male","1234");
               Person employee = new Employee("Rwagaju","Female","4456");
               student.work();
               employee.work();
               //using methods implemented in abstract class - inheritance
               employee.changeName("Gervais");
               System.out.println(employee.toString());
     }

}
```

II.2. What is an interface in Java ?

Interface is the best way to achieve abstraction in java. Java interface is also used to define the **contract** for the sub classes to implement. Contract means all methods have to be implemented.

For example, let’s say we want to create a drawing consists of multiple shapes. Here we can create an interface **Shape** and define all the methods that different types of Shape objects will implement. For simplicity purpose, we can keep only two methods:

draw() to draw the shape and **getArea()** that will return the area of the shape.

Shape Interface above requirements.

File name :**Shape.java**

```
public interface Shape {

    //implicitly public, static and final variables
    public String LABEL="Shape";

    //interface methods are implicitly abstract and public
    void draw();

    double getArea();
}
```

Important Points about Interface in Java

1. The **interface** is the keyword that is used to create an interface in java.
2. We can’t instantiate an interface in java.
3. Interface provides absolute abstraction, as abstract classes in java to but abstract classes can have method implementations but interface can’t.
4. Interfaces can’t have constructors because we can’t instantiate them and interfaces can’t have a method with body.
5. By default any attribute of interface is **public**, **static** and **final**, so we don’t need to provide access modifiers to the attributes but if we do, compiler doesn’t complain about it either.
6. By default interface methods are implicitly **abstract** and **public**, it makes total sense because the method don’t have body and so that subclasses can provide the method implementation.
7. An interface can’t **extend** any class but it can **extend** another **interface**.
8. The **implements** keyword is used by classes to implement an interface.

9. A class implementing an interface must provide implementation for all of its method unless it’s an abstract class. For example, we can implement above interface in abstract class like this:

   ShapeAbs.java

```
   ublic abstract class ShapeAbs implements Shape {

@Override
public double getArea() {
        // TODO Auto-generated method stub
        return 0;
}

   }
```

10. We should always try to write programs in terms of interfaces rather than implementations so that we know beforehand that implementation classes will always provide the implementation and in future if any better implementation arrives, we can switch to that easily.

Java Interface Implementation Example:

File Name : Circle.java

```java
public class Circle implements Shape {

    private double radius;

    public Circle(double r){
            this.radius = r;
    }

    @Override
    public void draw() {
            System.out.println("I am Drawing a Circle of radius :"+this.radius);
    }

    @Override
    public double getArea(){
            return Math.PI*this.radius*this.radius;
    }

    public double getRadius(){
            return this.radius;
    }
}
```

Notice that Circle class has implemented all the methods defined in the **interface** and it has some of its own methods also like **getRadius**().

Difference between abstract class and interface

The Main difference is that methods of a Java interface are implicitly abstract and cannot have implementations but a Java abstract class can have instance methods that implements a default behavior.

1. Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
2. Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..
3. Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
4. An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
5. A Java class can implement multiple interfaces but it can extend only one abstract class.
6. Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.

7. In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.

Comparison Table

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have	Members of a Java interface are

class members like private, protected, etc.	public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Note: When we started **OOP**, we have said: "**A class is a blue print of an Object !**" now we add : "**and the interface is a blue print of a class !**" , end of OOP!

III. JAVA AND COLLECTIONS

In the past, Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack,** and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Like the way that you used **Vector** was different from the way that you used **Properties, Thus invention of the collection framework to meet:**

- High-performance in implementations for the fundamental collections such as : dynamic arrays, linked lists, trees, and hash tables
- Allow different types of collections to work in a similar manner and with a high degree of interchangeability.
- Extend and/or adapt a collection easily.

Towards this end, the entire collections framework is designed around a set of standard **interfaces**. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided and you may use **as-is** or implement your own collection, **if you decide so**.

The collections framework is a unified architecture for representing and manipulating collections of objects.

The collection framework is equipped with the following features :

- **Interfaces** :

  These are abstract data types that represent different collections. They are Interfaces that allow collections to be manipulated independently of the details of their representation.

- **Implementations, i.e., Classes:**

  These are the concrete implementations of the collection interfaces.

- **Algorithms:**

  The methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be **polymorphic**: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several **map** interfaces and classes. **Maps** store **key/value** pairs. Although maps are **not** ***collections*** in the proper use of the term, but they are fully integrated with collections to provides an architecture to store and manipulate the group of objects in general as all collections does.

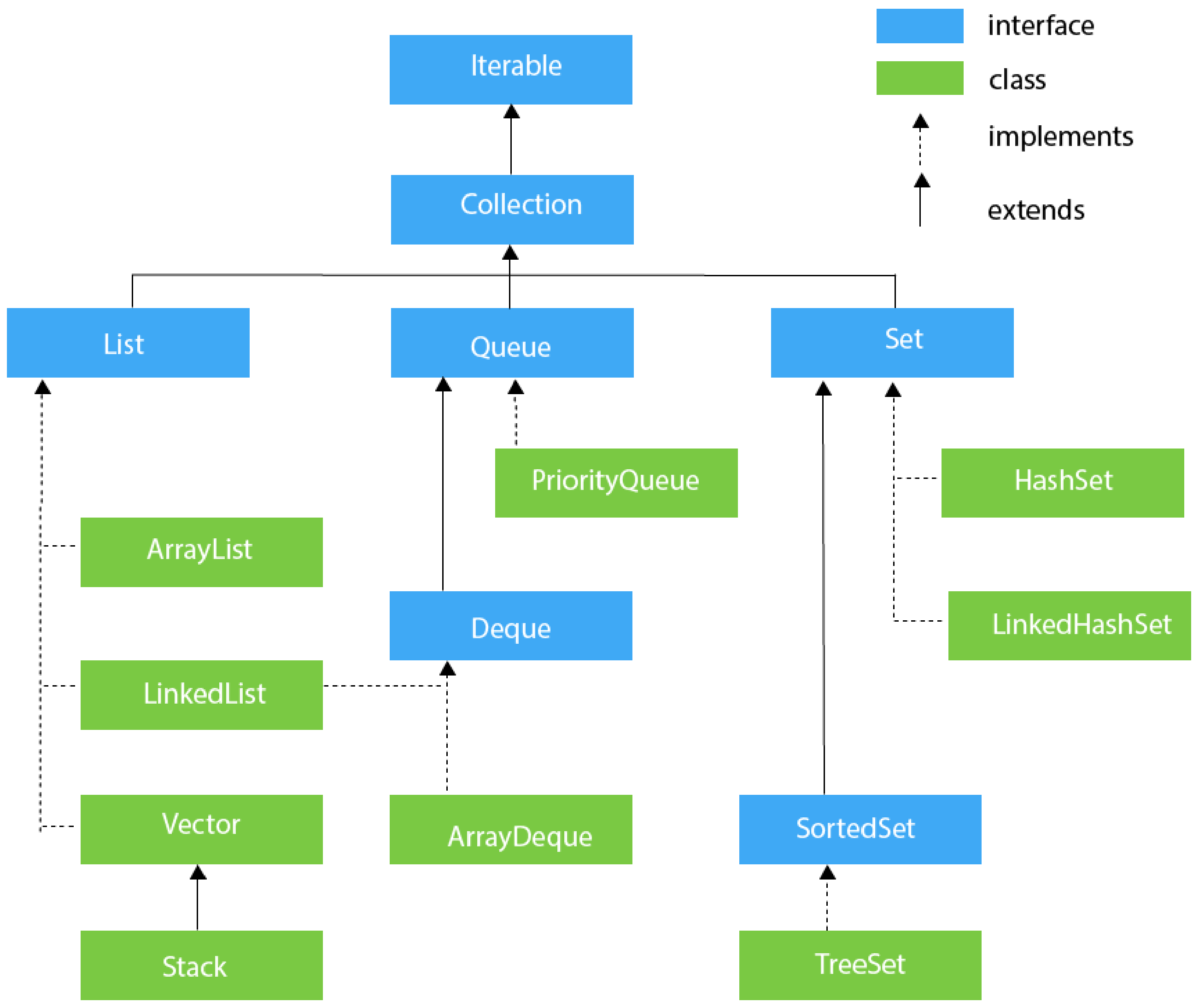
Java Collections can achieve all the operations that you perform on a data such as **searching**, **sorting**, **insertion**, **manipulation**, and **deletion**.

Characteristics of a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

Hierarchy of Collection Framework

The **java.util** package contains all the classes and interfaces for the Collection framework.



The Collection Interfaces

Sr.No.	Interface & Description
1	The Collection Interface This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	The List Interface This extends **Collection** and an instance of List stores an ordered collection of elements.
3	The Set This extends Collection to handle sets, which must contain unique elements.
4	The SortedSet

	This extends Set to handle sorted sets.
5	The Map This maps unique keys to values.
6	The Map.Entry This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	The SortedMap This extends Map so that the keys are maintained in an ascending order.
8	The Enumeration This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used **as-is** and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

Sr.No.	Class & Description
1	**AbstractCollection** Implements most of the Collection interface.
2	**AbstractList** Extends AbstractCollection and implements most of the List interface.
3	**AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	**LinkedList** Implements a linked list by extending AbstractSequentialList.
5	ArrayList Implements a dynamic array by extending AbstractList.
6	**AbstractSet**

	Extends AbstractCollection and implements most of the Set interface.
7	HashSet Extends AbstractSet for use with a hash table.
8	LinkedHashSet Extends HashSet to allow insertion-order iterations.
9	TreeSet Implements a set stored in a tree. Extends AbstractSet.
10	**AbstractMap** Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap

	Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents.

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

Sr.No.	**Class & Description**
1	Vector This implements a dynamic array. It is similar to ArrayList, but with some differences.
2	Stack Stack is a subclass of Vector that implements a standard last-in, first-out stack.

	Dictionary
3	Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
	Hashtable
4	Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
	Properties
5	Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
	BitSet
6	A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

The Collection Algorithms

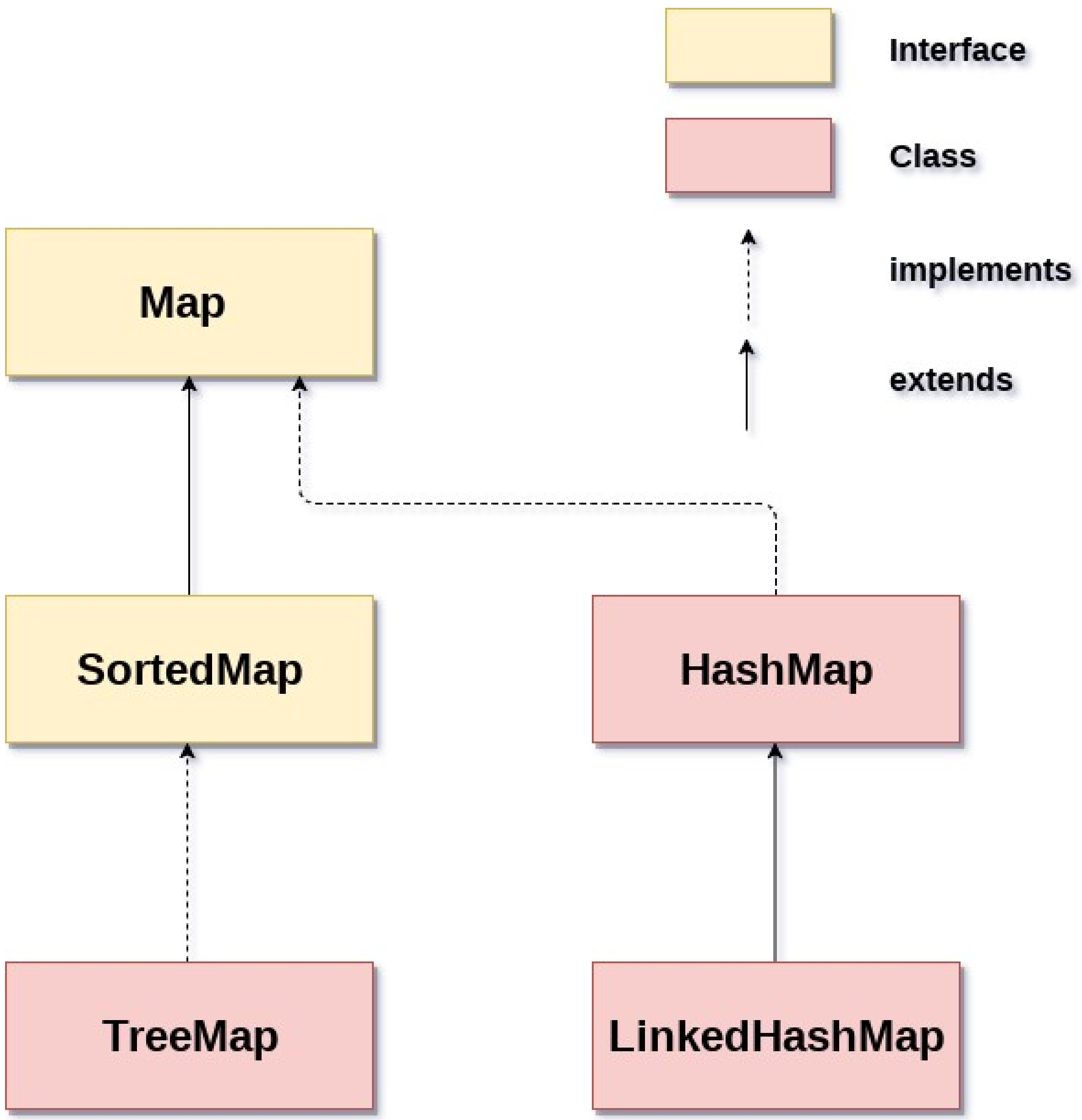
The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an

UnsupportedOperationException, which occurs when an attempt is made to modify an unmodifiable collection.

Map Interface in Java

The Map interface present in java.util package represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

them.

Summary

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();