Topic: Multiple Inheritance.
OOP concepts involved: Objects, Inheritance, Classes, Abstract Classes, Interfaces, Polymorphism, Abstraction.
Programming generic concepts involved: Functions, Variables, Data Types, Parameters, Constants, Access Modifiers.

➢ Theoric introduction

MULTIPLE INHERITANCE

Multiple inheritance in **Java** is the capability of creating a single class with multiple superclasses. Unlike some other popular object-oriented programming languages like C++, Java doesn't provide support for multiple inheritance in classes.

One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state* (which is the ability to inherit fields from multiple classes) and *multiple inheritance of implementation* (which is the ability to inherit method definitions from multiple classes).

The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

The **Java** programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements.

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Note: As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

WHAT IS AN INTERFACE?

An interface is a reference type in Java. It is similar to class, but the body of an interface can include only abstract methods and final fields (constants). A class implements an interface by providing code for each method declared by the interface.

The Java compiler adds ***public*** and ***abstract*** keywords by default before the interface method. Moreover, it adds ***public***, ***static*** and ***final*** keywords by default before data members.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class needs to implement. Generally speaking, ***interfaces*** are such contracts.

SINTAX FOR INTERFACE DECLARATION

```
interface <interface_name> {
    // declare constant fields
    // declare static and default methods
    // declare methods that are abstract
    // by default.
}
```

An ***interface*** is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a .class file.

- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Here's a basic interface that defines a single method, named ***Playable***, that includes a single method named ***play***:

```
public interface Playable
{
    void play();          // method is public abstract by default
}
```

This interface declares that any class that implements the ***Playable*** interface must provide an implementation for a method named **play** that accepts no parameters and doesn't return a value.

Notice that the name of the interface (***Playable***) is an adjective. Most interfaces are named with adjectives rather than nouns because they describe some additional capability or quality of the classes that implement the interface. Thus, classes that implement the ***Playable*** interface represent objects that can be played.

IMPLEMENTATION OF AN INTERFACE

To implement an interface, a class must do two things:

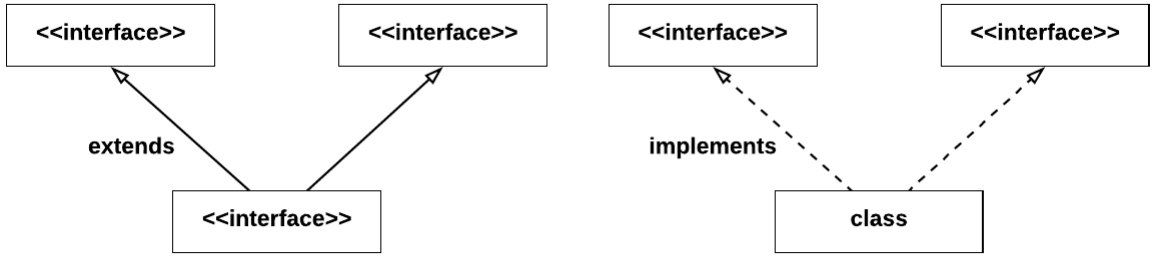
- It must specify an **implements** clause on its class declaration.
- It must provide an implementation for every method declared by the interface (unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class).

Here's a class that **implements** the *Playable* interface:

```
public class TicTacToe implements Playable
{
    // additional fields and methods go here
    public void play()
    {
       // own implementation goes here (code that plays the game)
    }
    // additional fields and methods go here
}
```

Here, the declaration for the TicTacToe class specifies implements Playable. Then the body of the class includes an implementation of the play method.

IMPORTANT INTERFACE RULES



- You cannot instantiate an interface.
- An interface can extend multiple interfaces.
- An interface is not extended by a class; it is implemented by a class.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface does not contain any constructors.

DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASS

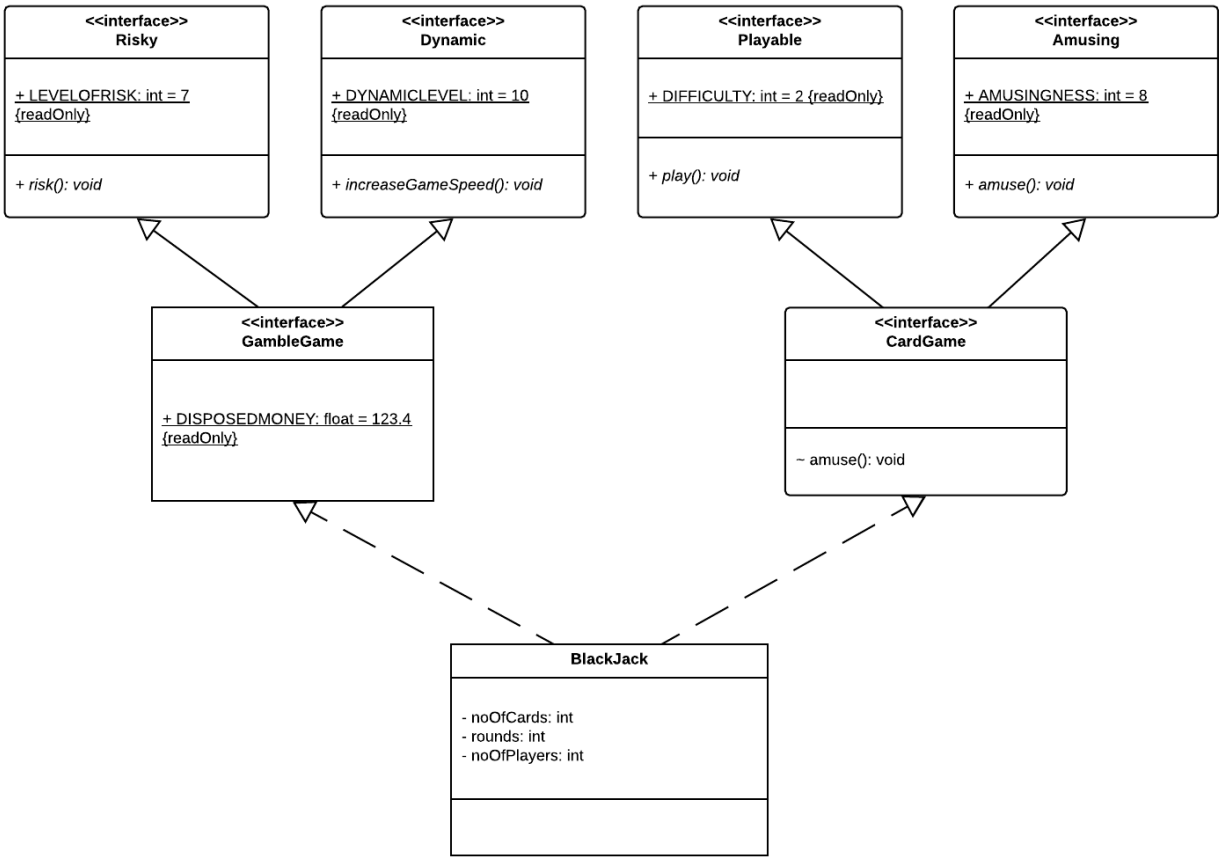
Interface	abstract Class
All the methods inside an interface are by default abstract.	An abstract class may or may not have abstract method.
You cannot define constructor in an interface.	An abstract class can contain constructor.
An interface can only have constants.	In an abstract class, the data variable may or may not be defined constant.
It supports multiple inheritance.	It supports single level inheritance.

➢ Statement

Using the concepts of multiple inheritance learned in the previous introduction, make a *GambleGame* interface that extends from a *Risky* named interface and from a *Dynamic* named interface. After that, make another interface named *CardGame* that extends from an Amusing interface and also from a *Playable* interface.

In the end, create a class named *BlackJack* with a couple of attributes such as **noOfCard**, **noOfPlayers** and **rounds**, that implements the *CardGame* and the *GambleGame* interfaces. Make sure of overriding the abstract methods provided by the interfaces.

➢ Class design (UML)



➢ **Program Code**

Risky.java

```
public interface Risky {
	int levelOfRisk = 7;	// 7 from 10

	void risk();
}
```

Dynamic.java

```
public interface Dynamic {
	int dynamicLevel = 10; // 10 out of 10

	void increaseGameSpeed();
}
```

GambleGame.java

```
public interface GambleGame extends Risky, Dynamic{
	float disposedMoney = 123.4f;
}
```

Playable.java

```
public interface Playable {
	int difficulty = 2; // 2 is medium, 3 is hard

	void play();
}
```

Amusing.java

```
public interface Amusing {
	// public static final by default
	int amusingness = 8;	// 8 out of 10

	//public abstract by default
	void amuse();
}
```

CardGame.java

```
public interface CardGame extends Playable, Amusing{
	// we are overwriting the Amusing interface method "amuse'
	@Override
	default void amuse() {
		System.out.println("This game is so fun to play. "+amusingness+" out
of 10!");
	}
}
```

BlackJack.java

```
public class BlackJack implements CardGame, GambleGame {

	private int noOfCards;
	private int rounds;
	private int noOfPlayers;

	public BlackJack(int noOfCards, int rounds, int noOfPlayers) {
		this.noOfCards = noOfCards;
		this.rounds = rounds;
		this.noOfPlayers = noOfPlayers;
	}

	public int getNoOfCards() {
		return noOfCards;
	}

	public void setNoOfCards(int noOfCards) {
		this.noOfCards = noOfCards;
	}

	public int getRounds() {
		return rounds;
	}

	public void setRounds(int rounds) {
		this.rounds = rounds;
	}

	public int getNoOfPlayers() {
		return noOfPlayers;
	}

	public void setNoOfPlayers(int noOfPlayers) {
		this.noOfPlayers = noOfPlayers;
```

```
	}

	// overriding interface methods, implementing them
	@Override
	public void play() {
		System.out.println(noOfPlayers + " individuals are playing BlackJack
with a total of " + noOfCards + " cards, "
				+ rounds + " rounds and a difficulty of " + difficulty);
	}

	@Override
	public void risk() {
		System.out.println("Each of the "+noOfPlayers+" players dispose of
"+disposedMoney+" dollars for playing at a level of risk of: "+levelOfRisk);
	}

	@Override
	public void increaseGameSpeed() {
		System.out.println("The game has increased its speed to a dynamic role
of: "+dynamicLevel);
	}
}
```

Main.java

```
public class Main {
	public static void main(String[] args) {
		BlackJack blackJack = new BlackJack(52, 6, 4);

		blackJack.amuse();
		blackJack.increaseGameSpeed();
		blackJack.play();
		blackJack.risk();

	}
}
```

- **Program execution**

In this code example, we created a set of interfaces, from which the ***Risky*** and ***Dynamic*** interfaces were extended through the ***GambleGame*** interface, we can see in this case, that we are applying the concept of **multiple inheritance**. The same happened with the ***CardGame*** interface, which

inherited from the Playable and Amusing interfaces. Each of the parent interfaces had a public abstract method and also a public static final variable (constant variable).

When the ***CardGame*** interface extended from the ***Amusing*** interface, it inherited an abstract method (**amuse**), which was overridden with an implementation of its own. The implementation consisted only of a System.out.println() method using the **amusingness** constant variable within it. The fact that we could make an actual implementation of the **amuse()** method is because we used the ***default*** access modifier.

So by the time the BlackJack class was extended by both GambleGame and CardGame, it was not necessary to implement the amuse() method overridden in the CardGame interface.

```
This game is so fun to play. 8 out of 10!
The game has increased its speed to a dynamic role of: 10
4 individuals are playing BlackJack with a total of 52 cards, 6 rounds and a difficulty of 2
Each of the 4 players dispose of 123.4 dollars for playing at a level of risk of: 7
```

➢ Conclusions

An interface is useful to handle a type of contract that our classes need to fulfill. This contract specifies that the abstract methods defined in the interface must be implemented within the classes that implement these interfaces.

As it was mentioned at the beginning of the document, Java does not allow multiple inheritance using classes, however, a java class can make use of multiple inheritance by implementing several interfaces. This is done this way to avoid problems like when more than one parent class has the same method with the same structure and, therefore, the compiler ends up having conflicts.