Topic: Multiple Inheritance.

OOP concepts involved: Objects, Inheritance, Classes, Abstract Classes, Interfaces, Polymorphism, Abstraction.

Programming generic concepts involved: Functions, Variables, Data Types, Parameters, Access Modifiers.

➢ Theoric introduction

MULTIPLE INHERITANCE

Multiple inheritance 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:

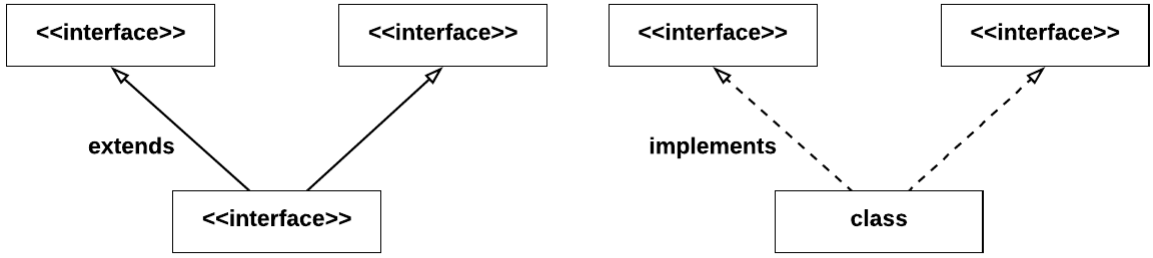
- It must specify an **implements** clause on its class declaration.
- It must provide an implementation for every method declared by the interface (unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class).

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.

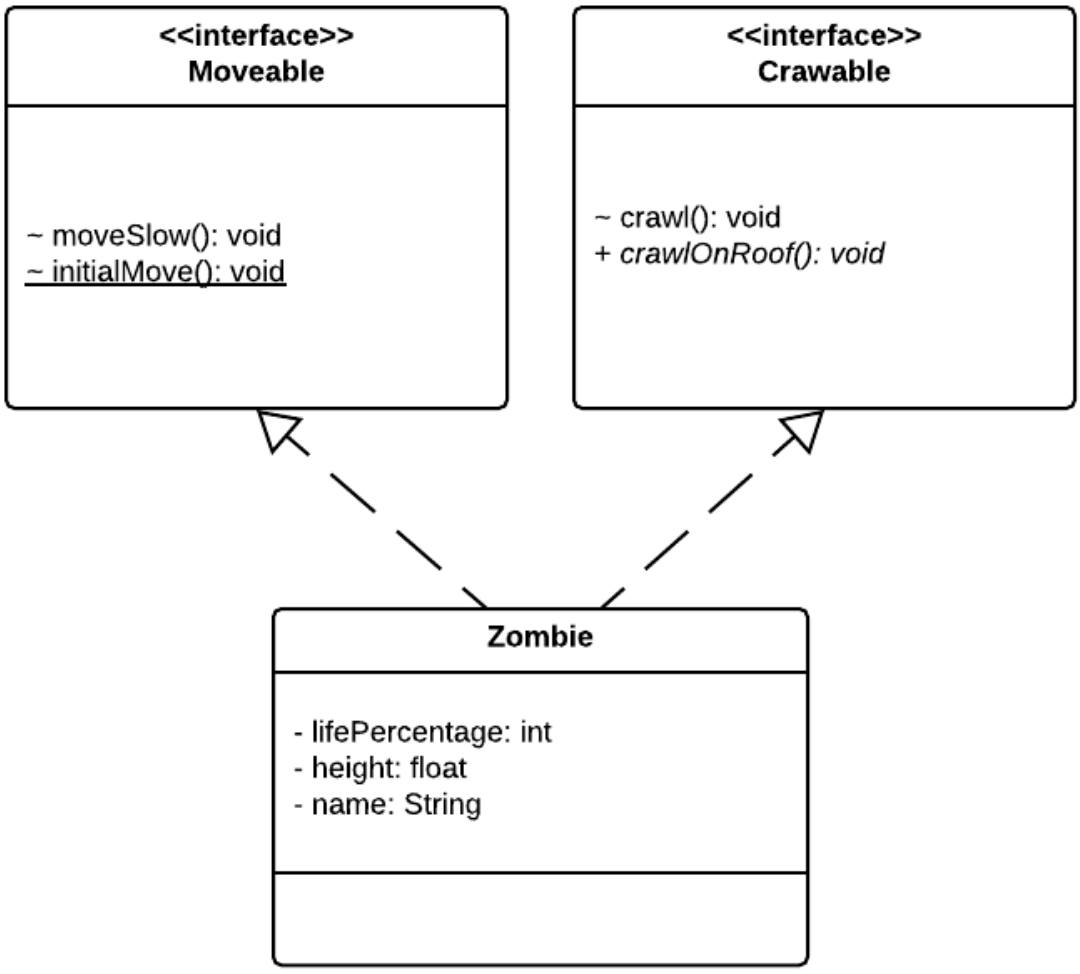
DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASS

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 class named Zombie that implements several interfaces such as Moveable and Crawable, each of them must have a static method and a default method specifying a certain implementation.

➢ **Class design (UML)**



➢ **Program Code**

Moveable.java

```
public interface Moveable {
	// we can add an implementation because we are using
	// "default" keyword
	default void moveSlow() {
		System.out.println("Be patient, I'm coooming fooor youuuu!!");
	}

	static void initialMove() {
		System.out.println("Let's get some braaaainsss");
	}
}
```

Crawable.java

```
public interface Crawable {
	// we can add an implementation because we are using
	// "default" keyword
	default void crawl() {
		System.out.println("Arghhhh, crooawwwliing!!");
	}

	// public abstract method by default
	void crawlOnRoof();
}
```

Zombie.java

```
public class Zombie implements Moveable, Crawable{

	private int lifePercentage;
	private float height;
	private String name;

	public Zombie(int lifePercentage, float height, String name) {

		this.lifePercentage = lifePercentage;
		this.height = height;
		this.name = name;

		Moveable.initialMove();
	}
```

```
	@Override
	public void crawlOnRoof() {
		System.out.println(name+" is crawling on the roof!");
	}

	public int getLifePercentage() {
		return lifePercentage;
	}

	public void setLifePercentage(int lifePercentage) {
		this.lifePercentage = lifePercentage;
	}

	public float getHeight() {
		return height;
	}

	public void setHeight(float height) {
		this.height = height;
	}

	public String getName() {
		return name;
	}

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

}
```

Main.java

```
public class Main {

	public static void main(String[] args) {
		// instantiating a Zombie
		Zombie niceZombie = new Zombie(50, 1.75f, "Rasco");

		// calling the default methods
		niceZombie.crawl();
		niceZombie.moveSlow();

		// calling the own implementation of crawlOnRoof
		// method made it within Zombie class
```

```
            niceZombie.crawlOnRoof();
        }

}
```

➢ **Program execution**

In this code example, the Zombie class extended from the ***Crawable*** and ***Moveable*** interfaces.
The ***Moveable*** interface had two body methods, due to the access modifiers used in its implementation (***default*** and ***static***). So when implementing that interface in the ***Zombie*** class, it was not necessary to override any method.

The same happened with the ***Crawable*** interface, it had a ***default*** method (crawl) with an implemented body and also a public abstract method (crawlOnRoof) without implementation, due to that, our Zombie class only needed to override the crawOnRoof method with its own implementation.

```
Let's get some braaaainsss
Arghhhh, crooawwwliing!!
Be patient, I'm coooming fooor youuuu!!
Rasco is crawling on the roof!
```

➢ **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.