# Abstraction concept

# Interfaces and abstract classes

# Polymorphism

# Abstraction concept

- One of the four concepts in OOP
- Abstraction is to represent essential features of a system without getting involved in the complexity of the entire system
- It allows the user to hide non-essential details relevant to user
- It allows only to show the essential features of the object to the end user(programmer)
- In other sense we can say it deals with the outside view of an object (interface)

# Abstraction concept

*Every day in our life we use abstraction ignoring the details which don't concern us*

Example:

When using some device for memory storage (flash memory, hard disk, CD) we don't care how it works inside. We need to know only how to copy, paste and delete files on it.

# Problem

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts.

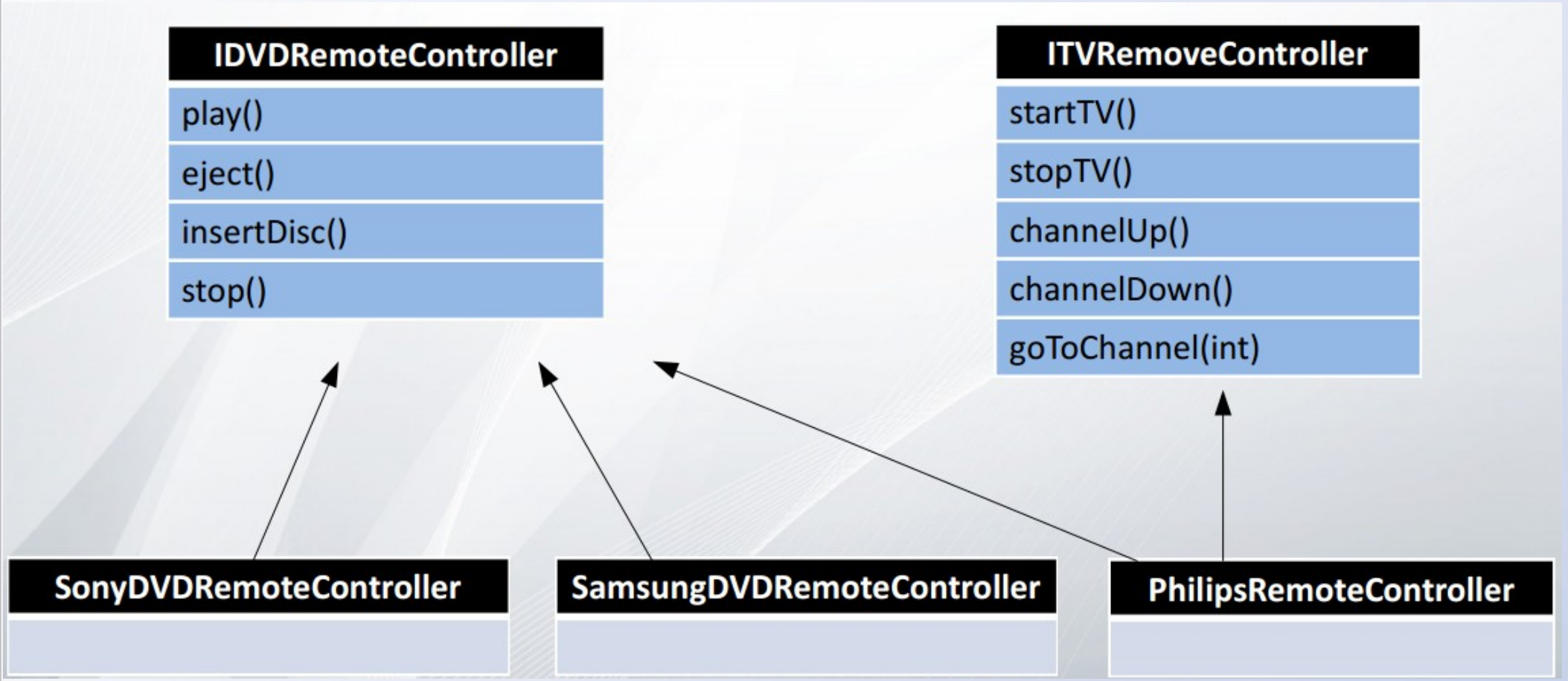
Each group should be able to write their code without any knowledge of how the other group's code is written.

Generally speaking, interfaces are such contracts.

# Interface

- An interface is a reference type, similar to a class but:
- It can contain only constants and method signatures
- There are no method bodies
- Interfaces cannot be instantiated—they can only be implemented by classes (or extended by other interfaces)
- It defines a “contract“ for behavior which the classes agree with
- Keyword *interface* is used for creating

# Remote controller example




IT TALENTS
Training Camp

# Interface

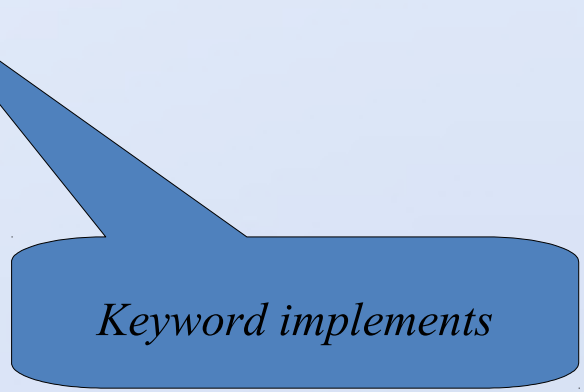
```
public interface IDVDRemoteController {
    void play();
    void eject();
    void insertDisc();
    void stop();
}
```

*Keyword interface*

```
public interface ITVRemoteController {
    void startTV();
    void stopTV();
    void channelUp();
    void channelDown();
    void goToChannel(int channelNumber);
}
```

# Implementing an interface

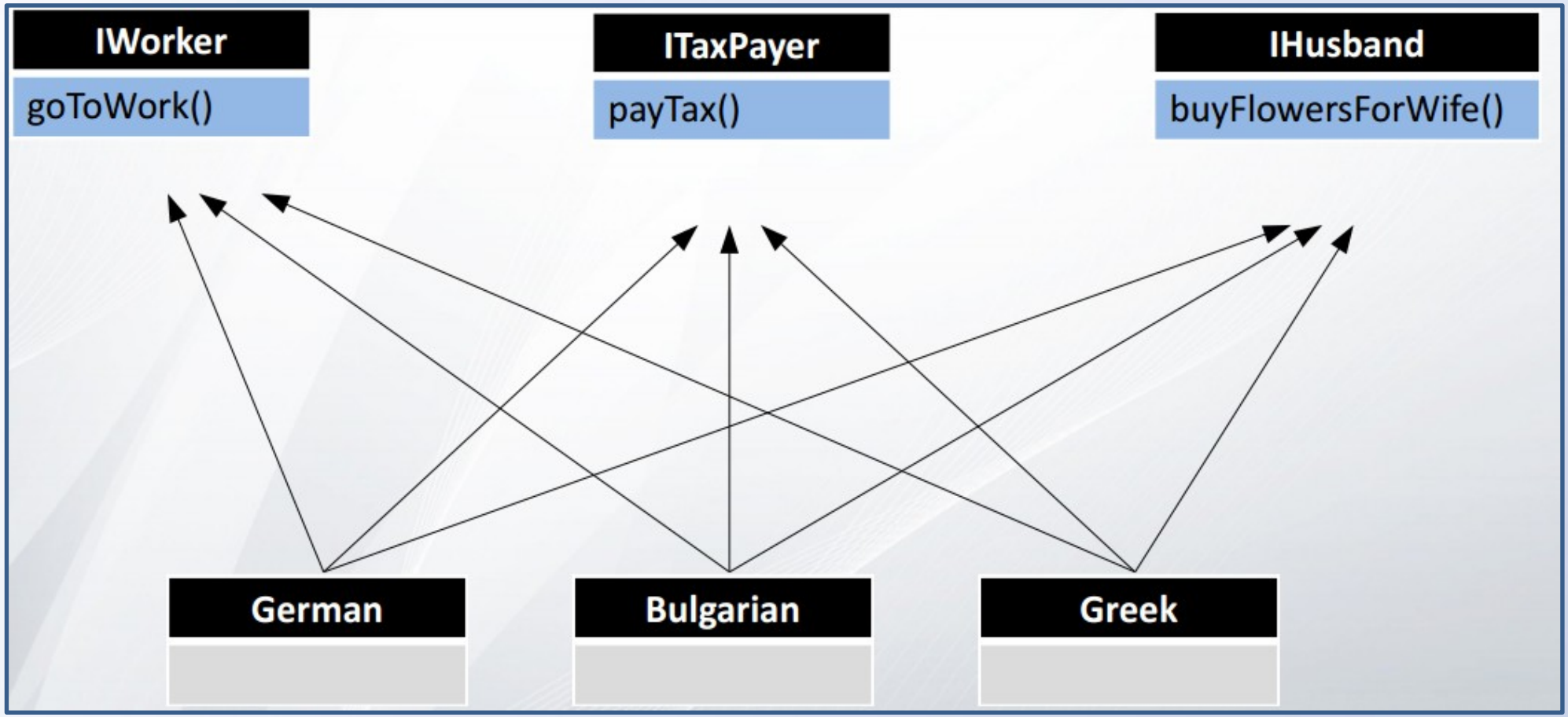
```
public class SamsungDVDRemoteController implements IDVDRemoteController {

    public void play() {
        System.out.println("Welcome to SAMSUNG DVD");
    }

    public void eject() {
        System.out.println("Eject...");
    }

    public void insertDisc() {
        System.out.println("Eject...");
    }

    public void stop() {
        System.out.println("Stop movie...");
    }
}
```



# More about iterfaces and their implementations

- All methods in the interface have *public* access no matter if this is implicitly set
- A class that implements an interface must implements all the methods in the interface (or it should be declared as abstract)
- A class that implements an interface cannot reduce the visibility of the methods in the interface
- An interface can *extends* another interface (but this is rarely used)
- Only *public* and *default* modifiers are allowed for an interface

# Another example



# Method signature

- The method signature contains his name, plus the number and the type of its parameters(in the same order)
- Method's return type is not part of the signature
- A class or interface shouldn't have two methods with the same signature
- When you implement method from an interface or override method in subclass you cannot change its return type

# Abstract class

When some class “is not complete” because cannot describe all the behavior needed to do what its supposed to do, then it should be declared as abstract

For instance, we want to modify the class Animal and add method makeSomeNoise() because each animal can emit some typical sound.

Class Animal is unable to create a meaningful implementation for this method because different animals emit different sounds.

# Abstract class

- Abstract classes define functionality which is not completed
- Abstract method is a method with definition but without implementation
- Abstract classes may contain abstract methods
- Class with at least one abstract method should be declared as abstract
- Abstract classes cannot be instantiated

# Animal example

*Keyword abstract*

```
public abstract class Animal {
    private int age;
    private double weight;

    public void breathe() {
        System.out.println("Breathing...");
    }

    public void walk() {
        System.out.println("Walking...");
    }

    public abstract void makeSomeNoise();
}
```

*abstract method has no body*

# Animal example

```
public class Cat extends Animal{
    void climb() {
        System.out.println("Climbing...");
    }
    public void makeSomeNoise() {
        System.out.println("Myal myal...");
    }
}
```

*Implementation of the abstract method from the parent class*

```
public class Dog extends Animal {
    boolean isDangerous;
    void bringStick() {
        System.out.println("Bringing the stick...");
    }
    public void makeSomeNoise() {
        System.out.println("Bau bau...");
    }
}
```

*Implementation of the abstract method from the parent class*

# Implementing an abstract method in the subclass

- Class which extends an abstract class should implement (override) all of the abstract methods from the parent class
- Otherwise, the subclass also should be declared as abstract
- Pure abstract class is abstract class with no fields and no concrete methods(it other words it contains only abstract methods)
- Pure abstract class is almost the same as interface. The difference is that a class can implement many interfaces but can extend only one class

# Zoo example

Let's create class Zoo representing zoo with animals

- The class holds array with Animal
- The class has method addAnimal which adds animal to the zoo

*Keep encapsulation concept in the class!*

# Zoo example

```
public class Zoo {
    private Animal[] animals;

    public Zoo(int cages) {
        animals = new Animal[cages];
    }

    public void addAnimal(Animal newAnimal) {
        for (int i = 0; i < animals.length; i++) {
            if(animals[i] == null) {
                animals[i] = newAnimal;
                return;
            }
        }
        System.out.println("No free cages for more animals!");
    }

    public Animal[] getAnimals() {
        return animals;
    }
}
```

# Zoo example

- There is no problem to declare array of Animal (Animal is abstract class)

```
private Animal[] animals;
```

- Also, there is no problem to declare variable, field or argument which is interface or abstract class
- A reference of interface type can be initialized with instance of class which implements this interface
- A reference of some type can be initialized with instance of any type which extends the type of the reference

# Zoo example

Let's create class ZooDemo with main method in it

- Create an instance of Zoo
- Create one instance for the classes Cat, Dog and Bird
- Try to declare some of them as type Animal
- Add them to the zoo using method addAnimal

# Zoo example

As you see, it's ok to pass instance of Dog or Bird although the method addAnimal has argument of type Animal

```
public class ZooDemo {
    public static void main(String[] args) {
        Zoo zoo = new Zoo(10);
        Animal cat = new Cat();
        Dog dog = new Dog();
        Bird bird = new Bird();

        zoo.addAnimal(cat);
        zoo.addAnimal(dog);
        zoo.addAnimal(bird);
    }
}
```

*No problem to pass Dog although the argument is of type Animal, because Dog extends Animal*

# What`s new in Java 8 ?

- Default methods – a functionallity that  provides flexibility to allow interface define implementation which will use as default in the situation where a concrete class fails to provide an implementation for that method.

```
public interface IRobot
{
    void doSomething();

    default void shootWithGun()
    {
        System.out.println("Chappy no shoot, Chappy wants peace!");
    }


}
```

# Why Default methods ?

- Reengineering an existing JDK framework is always very complex. Modify one interface in JDK framework breaks all classes that extends the interface which means that adding any new method could break millions of lines of code. Therefore, default methods have introduced as a mechanism to extending interfaces in a backward compatible way.

- Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. If each added method in an interface defined with implementation then no implementing class is affected. An implementing class can override the default implementation provided by the interface.

# More on default methods

- http://java.dzone.com/articles/interface-default-methods-java