OOP Best practices: Interfaces, abstract classes, polymorphism

IT Learning &
Outsourcing Center

www.pragmatic.bg

Lector: Stefan Vadev
Skype: stefanvadev77
E-mail: stefan.vadev@gmail.com
Facebook:
https://www.facebook.com/stefan.vadev.7

Summary

- Abstraction concept
- Interfaces and abstract classes
- Polymorphism

Recap

- Inheritance
- Constructors - super

Quick Intro to Arrays

- `int[] numbers = new int[10];`
- `String[] names = new String[5];`

Quick Intro toArrays

- names[0] = “Pancho”;
- names[1] = “Stavri”;

Quick Intro to Arrays


```
for (int i = 0; i < names.length; i++) {
String name = names[i];
System.out.println(name);
}
```

- Be careful of ArrayIndexOutOfBoundsException

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
- 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.

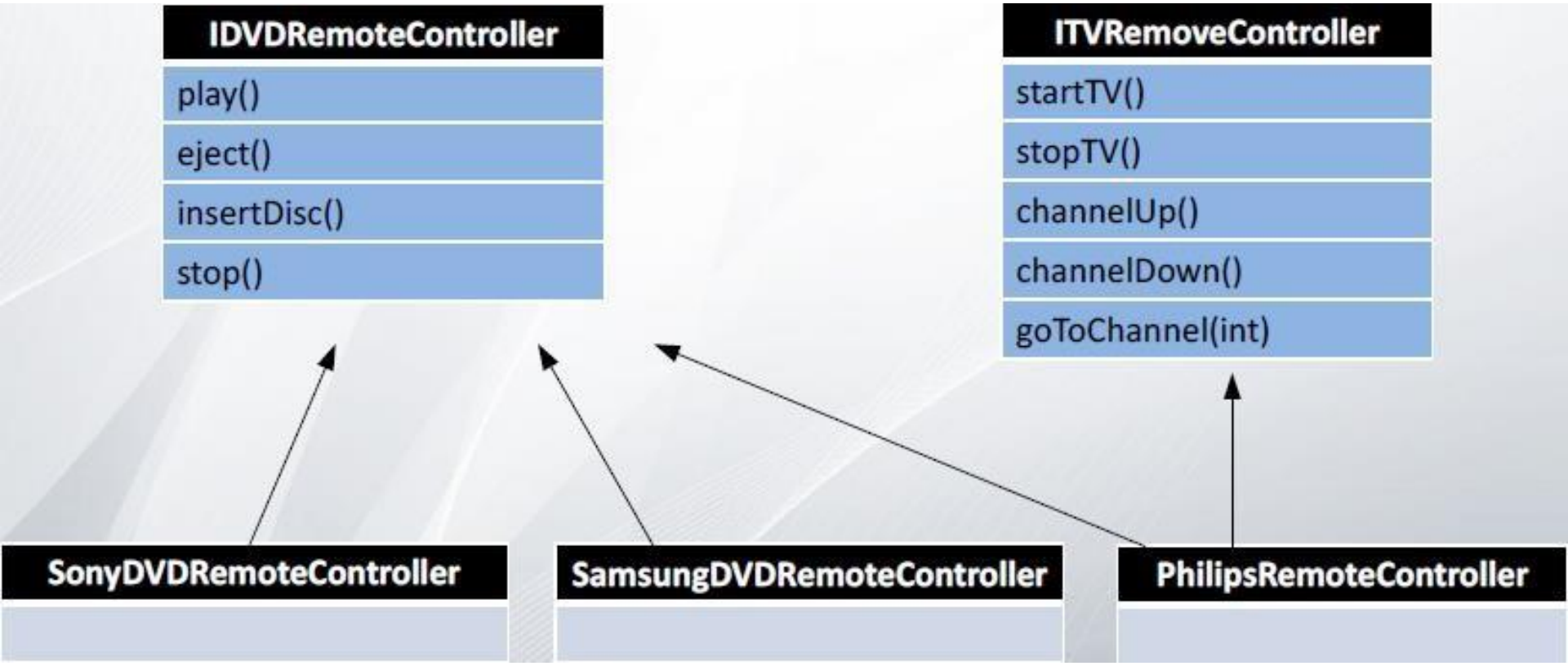
Problem

- 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 (prior to Java8)
  - 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

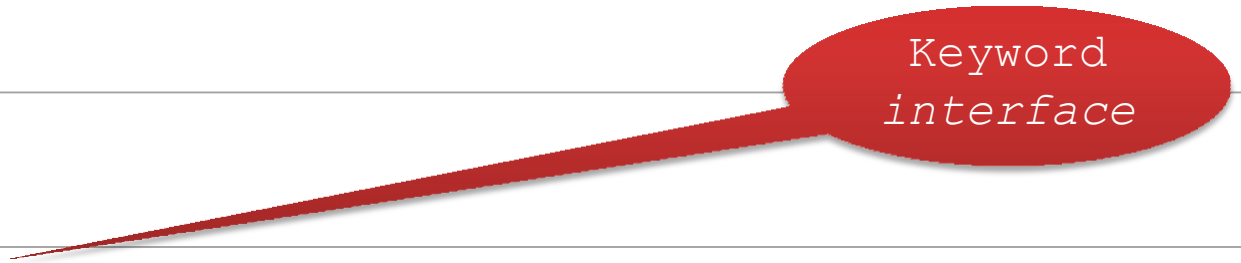
Remote controller example



- One class can extends only one class, but can implement many interfaces

Interface


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




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

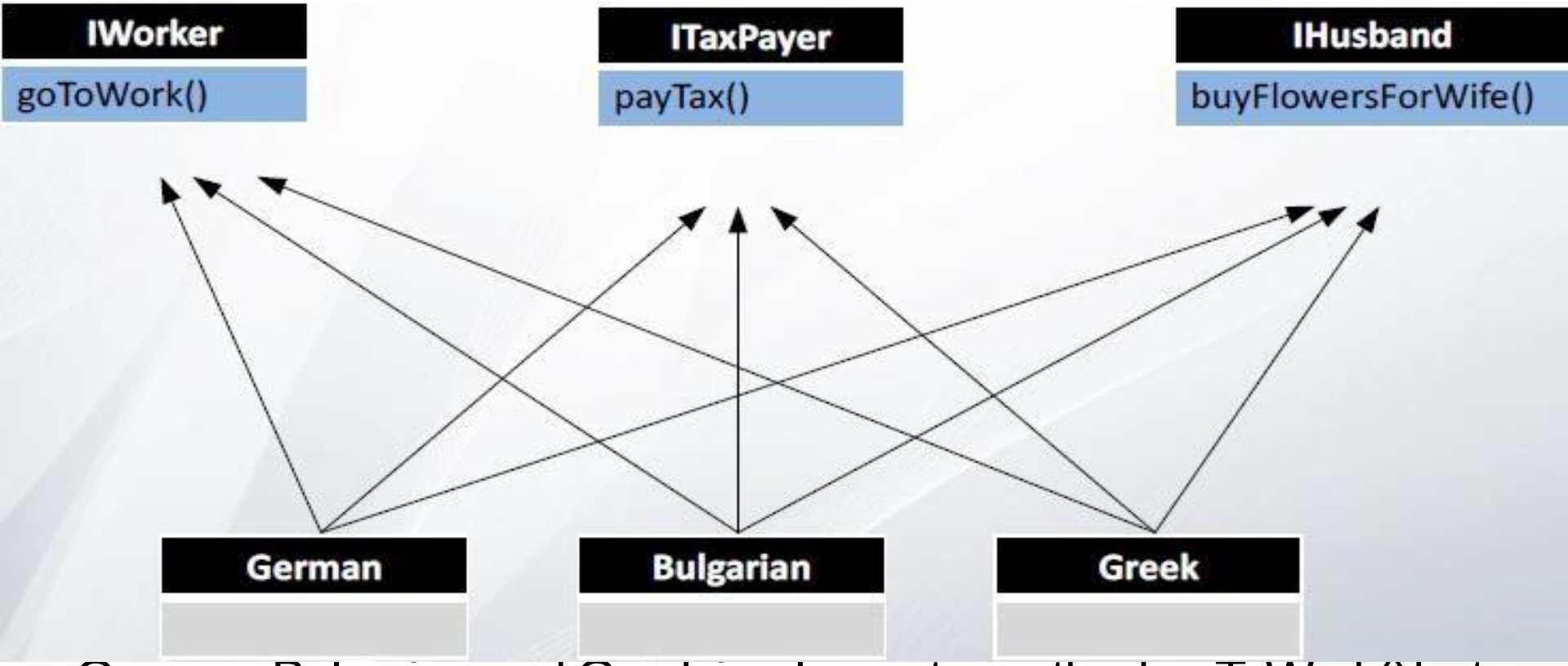
Implementing an interface


```
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*
- Only *public* and *default* modifiers are allowed for an interface

Another example



- German, Bulgarian and Greek implements method goToWork() but everyone in a different way:
- The German is always on time, the Bulgarian goes to work everyday and the Greek goes to work when he wants

Method signature

- The method signature contains its 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 produce different sounds.

Abstract class

- Abstract class define functionality which is not completed
- Abstract method is 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 {
        protected int age;
        protected 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 parent class

```
public class Dog extends Animal{
        void climb() {
            System.out.println("Climbing...");
        }

        public void makeSomeNoise() {
            System.out.println("Bau bau...");
        }
}
```

Implementation of the abstract method from 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 implements many interfaces but 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

Polymorphism

- Polymorphism is one of the four concepts in OOP
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- In other word, a variable with a given name may be allowed to have different forms and the program can determine which form of the variable to use at the time of execution
- In java polymorphism is achieved by overriding methods in the subclass
- Polymorphism is a generic term that means 'many shapes'

Demonstrating polymorphism example

- Let's override method walk() in classes
  - Dog, Cat and Bird

- This way, each animal will walk in different way, typical for the respective animal

Demonstrating polymorphism example


```
public class Cat extends Animal{
        ...
        @Override
        public void walk() {
          System.out.println("Walking like a cat");
        }
}
```

```
public class Dog extends Animal{
        ...
        @Override
        public void walk() {
           System.out.println("Walking like a dog");
        }
}
```

Demonstrating polymorphism example

- What happens when in the main method in class ZooDemo try to invoke method walk for the object cat?

```
public class ZooDemo {
	public static void main(String[] args) {
	...

	Animal cat = new Cat();

	...
	cat.walk();
	}
}
```

Demonstrating polymorphism example

- The output in the console is:
  - Walking like a cat
- The method walk() from class Cat has been invoked instead of walk() from class Animal

- The programmer (and the program) does not always have to know the exact type of the object in advance, and so the exact behavior is determined at run-time (this is called late binding or dynamic binding)

Demonstrating polymorphism example

- You always deal with reference, but the method which will be called depends on the type of the instance, not the type of the reference
- In some other OOP languages there is a term *virtual method* (or function)
- In java all methods are virtual, so every time you invoke a method, the decision which method to be called is taken at runtime and it depends on the instance

Demonstrating polymorphism example

- Let's complete our example and call methods walk and makeSomeNoise for all the animals in the cage.

Demonstrating polymorphism example


```
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);

        Animal[] animalsInTheZoo = zoo.getAnimals();

        for (int i = 0; i < animalsInTheZoo.length; i++) {
                if(animalsInTheZoo[i] != null) {
                    animalsInTheZoo[i].walk();
                    animalsInTheZoo[i].makeSomeNoise();
                }
        }
}
```

More about references and instances

- Lets add method sing() in the class Bird

```
public class Bird extends Animal {
        ...
        public void sing() {
            System.out.println("Singing...");
        }
}
```

- What will happen when try to do this?

```
Animal bird = new Bird();
bird.sing();
```

More about references and instances

- This will result in a compilation error, because there is no such method declared in class Animal, although the instance is of type Bird.

- Remember:
- Which methods can be called depends on the reference type
- But which body will be executed depends on the instance type

What's the solution? Downcasting

- Downcasting (or just casting) is used to explicitly say to the compiler that reference of a base class refers to an instance of its subclass

```
public static void main(String[] args) {
        Animal bird = new Bird();
        ((Bird)bird).sing();
}
```

Downcasting (or just casting)

Downcasting

- Downcasting is unsafe operation and is good idea to check if the reference refer to an instance of the right class using *instanceof* operator

```
for (int i = 0; i < animalsInTheZoo.length; i++) {
        if(animalsInTheZoo[i] != null) {
            animalsInTheZoo[i].walk();
            animalsInTheZoo[i].makeSomeNoise();

            if(animalsInTheZoo[i] instanceof Bird) {
                Bird birdInZoo = (Bird) animalsInTheZoo[i];
                birdInZoo.sing();
            }
        }
}
```

Using *instanceof* before casting

Downcasting

- If we cast to wrong class we'll get exception of type
  - ClassCastException

```
zoo.addAnimal(cat);
zoo.addAnimal(dog);
zoo.addAnimal(bird);
Animal[] animalsInTheZoo = zoo.getAnimals();

for (int i = 0; i < animalsInTheZoo.length; i++) {
       if(animalsInTheZoo[i] != null) {
           animalsInTheZoo[i].walk();
           animalsInTheZoo[i].makeSomeNoise();
           Bird birdInZoo = (Bird) animalsInTheZoo[i];
           birdInZoo.sing();
       }
}
```

Upcasting

- Java permits an object of a subclass type to be treated as an object of any superclass type. This is called upcasting.
- Upcasting is done automatically, while downcasting must be manually done by the programmer
- Upcasting is a safe operation

Upcasting

- In most situations, the upcast is entirely unnecessary and has no effect. However, there are situations where the presence of the upcast changes the meaning of the statement(or expression).
- Suppose that we have overloaded methods:

```
public void doIt(Object o)...
public void doIt(String s)...
```

- If we have a String and we want to call the first overload rather than the second, we have to do this:

```
String arg = ...
doIt((Object) arg);
```

Final methods and classes

- *final keyword* can be applied to methods and classes
- Final class means that it cannot be extended by other class
- Final method means that this method cannot be overrided in the subclasses. This way its body is never changed

```
public final void walk() {
      System.out.println("Walking like a dog");
}
```

Summary

- What is abstraction and how to achieve it?
- What is interface? How to implement an interface?
- What is abstract class?
- What's the differences between interface and abstract class
- What is polymorphism and how to achieve it?
- Upcasting and downcasting
- What the meaning of *final* for methods and classes