Java Course
Lecture 4 - Static members, interfaces, abstract classes, polymorphism

PRAGMATIC IT Learning & Outsourcing Center

www.pragmatic.bg

Summary

- Static members
- Abstraction concept
- Interfaces and abstract classes
- Polymorphism

Static members

- Static members are class level methods or fields
- Static means ONE FOR ALL
- A static member is a member that can be used by itself, without reference to a specific instance
- You can declare both methods and variables to be **static**
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object

Static members

- Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.
- Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods.
  - They can only directly access **static** data.
  - They cannot refer to **this** or **super** in any way

Static constructor

- A static block is executed exactly once, when a class is first loaded (mentioned).

Example :

```
static {

}
```

Static Import

- By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface
- When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class

Inheritance

- What if we want to have a class which is exactly like another one but with some small change, maybe just one more method?
- Lets suppose we have a class Student and we want to create a second class CoolStudent which have the same fields like Student but with one addition "haircutStyle".

```
public class Student {
        int age;
        String name;
}

public class CoolStudent extends Student {
        String haircutStyle;
}
```

Inheritance(2)

- Inheritance help to limit and lower the duplications of code
- It helps to create hierarchy of objects using an "aggregation" relation
- Examples.. If we have 2D shapes like Rectangle, Square and so on we may create a basic class called Shape with methods like draw() or paint() or changeColor(), changeBorder() and so on.

So basic issues about this inheritance ?

- When we do inheritance for the child class ALL fields and methods of the parent (super) class should make sense.
- Liskov substitution principle

overriding

```
public class A {
	public void doSomething() {
		System.out.println(“doSomething in A”);
	}
}

public class B extends A {
	@Override
	public void doSomething() {
		System.out.println(“doSomething in B”);
	}
}
```

Problem when overriding

- Sometimes in an overridden method we may want to use the method of the parent

```
public class B extends A {
        @Override
        public void doSomething() {
                if(someCondition) {
                        System.out.println("doSomething in B");
                } else {
                        doSomething()??? Or this.doSomething?
                }
        }
}
```

Super keyword

- No the answer is super keyword

```
public class B extends A {
        @Override
        public void doSomething() {
                if(someCondition) {
                        System.out.println("doSomething in B");
                } else {
                        super.doSomething();
                }
        }
}
```

You can also use the super keyword to call constructor in the base class via super(…);

Multiple inheritance ?

- There is no multiple inheritance in Java.
- One class can extend 0 or 1 class but cannot extend more then one classes.
- By default if class doesn't extend anything it extends "Object" so this two are the same :

```
 class A {}
class A extends Object {}
```

Access Modifiers

- private (Most restrictive)
  - Only members of the same class can see this
- default
  - That's the access modifier used when no modifier is explicitly specified
  - Members of the same package can see default members
- protected
  - Only members of the same class, same package and children can see protected members
- public (No restrictions)
  - Everyone can see public members

Abstraction concept

- One of the 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.

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 behaviour 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 talk() 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 {
        private int age;
        private double weight;

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

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

        public abstract void talk();
}
```

Abstract method has no body

Animal example

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

	public void talk() {
	    System.out.println("Myal myal...");
	}
}
```

Implementation of the abstract method from parent class

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

	public void talk() {
	    System.out.println(“baf baf...");
	}
}
```

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

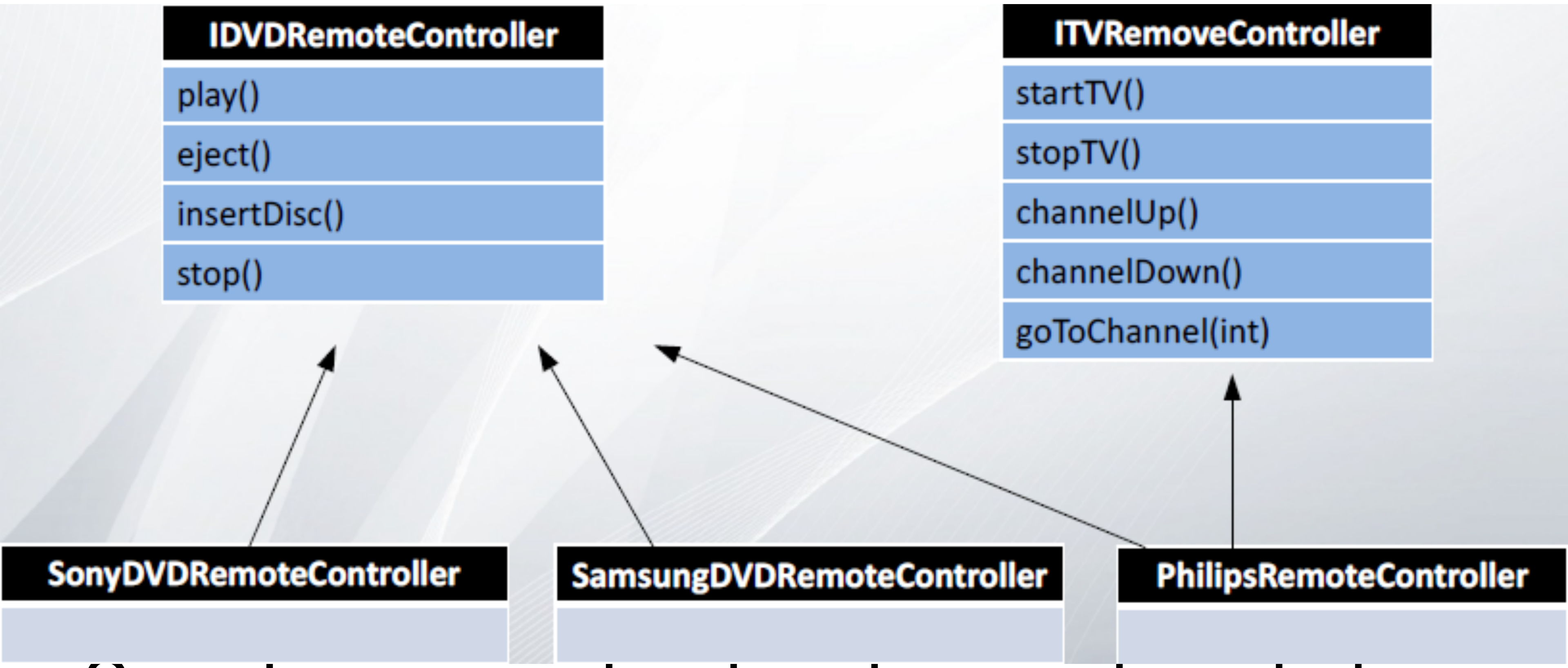
```
public class ZooDemo {
        public static void main(String[] args) {
            Zoo zoo = new Zoo(10);
            Animal cat = new Cat();
            Dog dog = new Dog();
            Bird bird = new Bird();

            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

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 (true only for java 7 and earlier). Java 8 introduces default methods and the ability to put static methods in interfaces.**
  - Interfaces cannot be instantiated—they can only be implemented by classes (or extended by other interfaces)
  - It defines a "contract" for behaviour which the classes agree with
  - Construction is similar to classes but instead of class we use the keyword interface

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();
}
```

Keyword *interface*

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

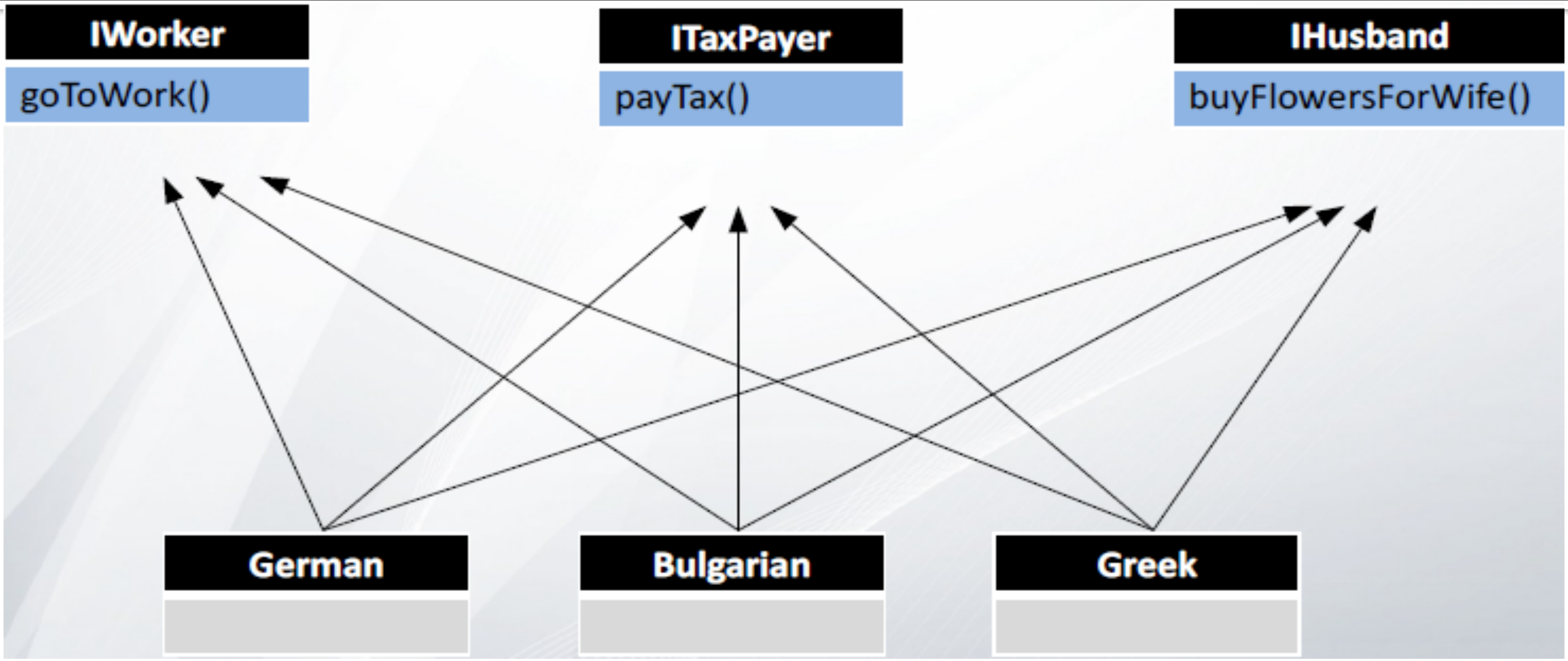
Implementing an interface

```
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 interfaces 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 *missing* modifiers are allowed for the interface itself

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

Polymorphism

- Polymorphism is one of the three 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 variable 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 behaviour 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
- 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 talk for all the animals in the cage.

Demonstrating polymorphism example

```java
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].talk();
                }
        }
}
```

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 you 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 not 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].talk();

          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].talk();
            Bird birdInZoo = (Bird) animalsInTheZoo[i];
            birdInZoo.sing();
        }
}
```

Result on next slide.

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 save operation
- Explicit upcasting is very rarely used

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);
```

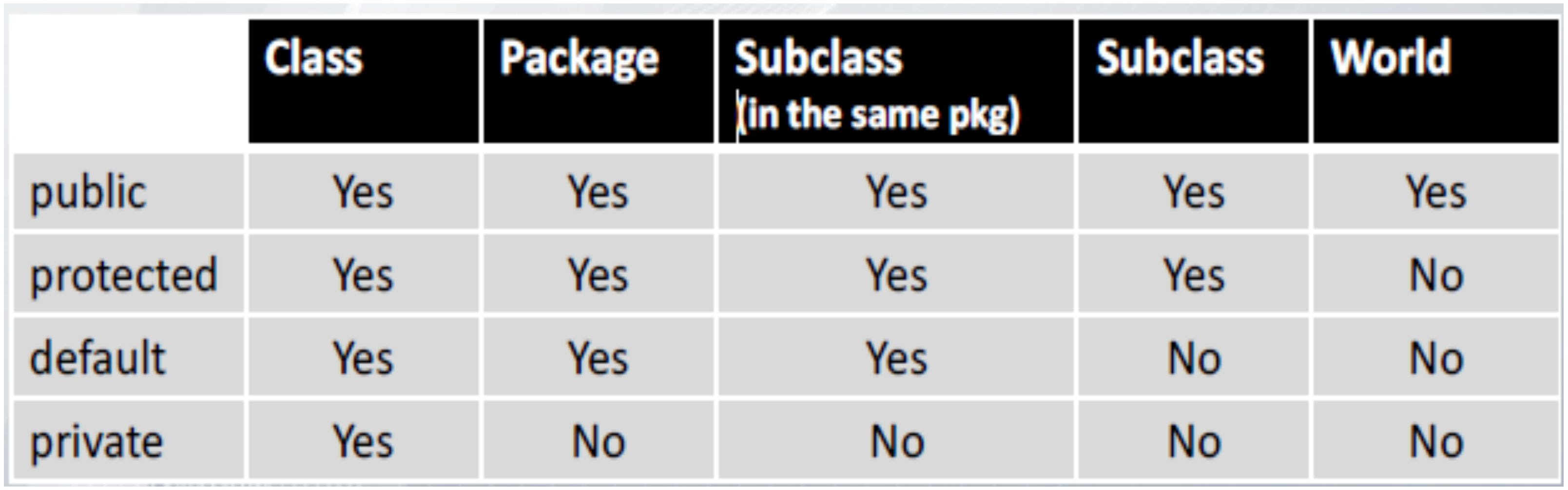
Class - Access modifiers

- Access modifiers are used to
  - Control access to classes (top level), methods, constructors or fields (bottom level) from outside the class
- For top level (classes) there are *public*, *default(package)* and in some cases *private(inner classes)*
- For bottom level: *public*, *protected*, *default and private*

Access modifiers in inheritance

- *public*, *private* and default(package)
- *protected* fields and methods is visible to the *classes* within the same package and in the child classes

Table with access levels:



	Class	Package	Subclass (in the same pkg)	Subclass	World
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

Explaining *public, private* and *default*

- *public* – gives access to the class, field or method from everywhere outside the class
- *private* – access is restricted only within the class
- *default/package* – visible from within the class and all other classes in the package
- *protected* – visible from within the class, all other classes in the package and all of the subclasses

Purpose of access modifiers

- Problem: If all fields of class Person are public they will be accessible from everywhere which evaluates the Encapsulation principle of OOP
- Accessibility directly to fields is dangerous and unsecure
- For accessing private fields outside the class are used public methods called „getter“ and „setter“

Getters and setters

- Getters are used for getting the value of private field outside the class.
- It should be implemented only if is necessary
- Setters are void methods and are used for setting the value of private field outside the class
- Validation can be implemented as part of the setter's body

Getter and setter example

```
private int age;

public int getAge() {
      return age;
}

public void setAge(int age) {
      if(age >= 0) {
            this.age = age;
      }
}
```

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");
}
```

Interfaces after Java 8 can have static methods

- You can declare static methods on interfaces:

```
public interface ContentFormatter {
    public void format();
    static String convertRegex(String
        regex) {
        ...
    }
}
```

- All the implementing classes will NOT have this method available to them
- The idea is : no more need for utility classes

Interfaces after Java 8 can have default methods

- Example:

```
public interface SomeInterface{
    public void normalMethod();
    default String otherMethodWithBody(String
        something) {
        ...
    }
}
```

- Default methods were added because of a “huge” functionality extension happened in Java 8 called Lambdas that we will speak about in later lections.

BIG note

- Default methods in interfaces doesn’t cause issues like the ones with “Multiple Inheritance” because:
  - If you implement two interfaces defining a default method with the same signature you will get compile time error and you have to override this method.
  - If you implement an interface with default method that extends another interface with default method having the same signature there is no issue we are receiving only the last one in the extending chain.

Static classes

- We may have a class inside another class

```
public class Test {

      public static class InternalExample {
            public void doSomething() {}
      }

}
```

- This is allowed however not very used. From the outside world we may use this class as follows:

```
Test.InternalExample internalStaticClass =

            new Test.InternalExample();
```

- Inside the class Test we may use just InternalExample

Internal classes

- We may have a class inside another class

```
public class Test {

        class InternalExample {
                public void doSomething() {}
        }

        public void doSomething () {}

}
```

- Cannot be instantiated from the outside world. Can be used only internally by class Test. Have access to "this" of the outer class. Have special syntax to do that for example to invoke method doSomething of the Test we may type "Test.this.doSomething()"

Local class

- Class inside a method

```
package test;

public class Test {

      protected void doSomething(){
            class A {
                  public void doSomethingA() {}
            }

            A a = new A();
            a.doSomethingA ();
      }
}
```

- Usually local classes implement some interface

Example sort students by score if same score by name

```
public class Student {
      int score;
      String name;
      public Student(int score, String name) {
            super();
            this.score = score;
            this.name = name;
      }
}
```

```java
public class Main {public static void main(String[] args) {
            Student[] students = new Student[3];
            students[0] = new Student(12, "ana");
            students[1] = new Student(15, "nay");
            students[2] = new Student(15, "nemo");
            class StudentComparator implements Comparator {
                  @Override
                  public int compare(Object o1, Object o2) {
                        Student s1 = (Student) o1;
                        Student s2 = (Student) o2;
                        if(s1.score > s2.score) {
                              return 1;
                        } else if (s1.score < s2.score){
                              return -1;
                        } else {
                           return s1.name.compareTo(s2.name);
                        }
                  }
            }
            Arrays.sort(students, new StudentComparator());
} }
```

Anonymous classes

- When we need something like Local class but only ones… there will be only one instance so we don't need to name it as "StudentComparator"

- So same example as before..We have the student array with students and we want to sort them :

Anonymous class example

```
Arrays.sort(students, new Comparator() {
      @Override
      public int compare(Object o1, Object o2) {
             Student s1 = (Student) o1;
             Student s2 = (Student) o2;
             if(s1.score > s2.score) {
                    return 1;
             } else if (s1.score < s2.score){
                    return -1;
             } else {
                return s1.name.compareTo(s2.name);
             }
      }
});
```

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

Problems?

- What's the difference between instance and static variables ?
- How many times is the static constructor executed?
- Can a static class member be accessed with an instance reference ?
- What are the different access modifiers ?
- What's the difference between protected and default access modifier ?
- What's a package ? What's a class's full name
- What's the difference between classes and abstract classes ?
- What's the difference between interfaces and abstract classes ?
- How many classes can a class have as a parent ?
- What's the difference between a final class and a normal one ?
- Can an abstract class be final, what about an interface ?