

# OOP

Avancerad Javaprogrammering

Utbildare: Mikael Olsson

# Dagens områden

- Inheritance
- Overriding
- Polymorphism
- Abstraction
- Encapsulation
- Interfaces

# Inheritance - Arv

- Klasser är lite som ritningar för objekt.
- Alla egenskaper och metoder som ingår för en klass ingår även för objekten vi skapar av denna klass.
- Arv innebär att vi kan låta en klass kopiera alla egenskaper och metoder från en annan klass.
- Barnklassen kan dessutom skriva över metoder/egenskaper och lägga till egna metoder/egenskaper.

# Inheritance - Extends

- Vi använder nyckelordet `extends` för att ärva från en klass.

```
class Super {  
    . . . . .  
}  
class Sub extends Super {  
    . . . . .  
}
```

# Inheritance - Exempel

```
class Calculation {
    int z;

    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the
given numbers:"+z);
    }

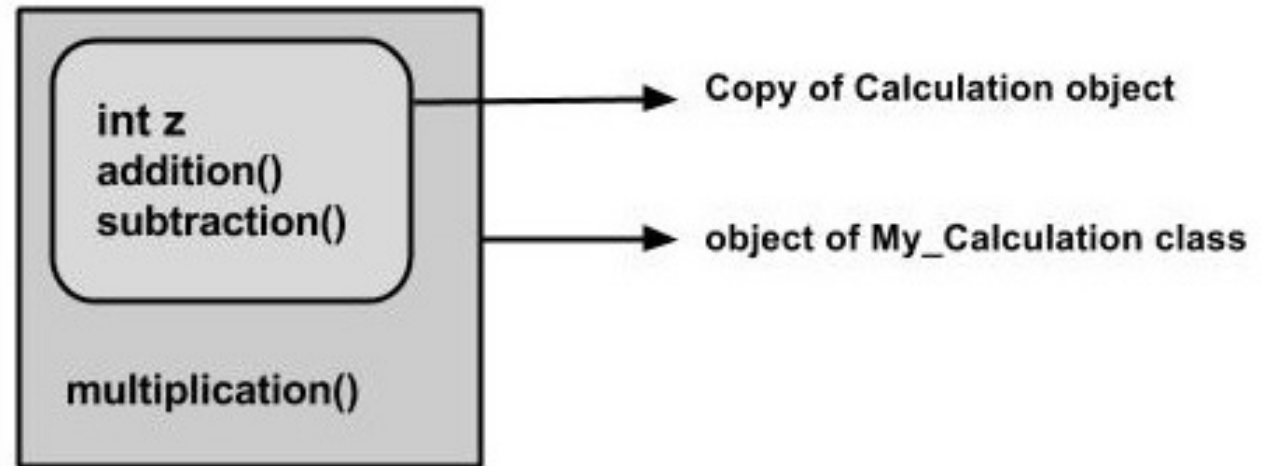
    public void Subtraction(int x, int y)
{
        z = x - y;
        System.out.println("The difference
between the given numbers:"+z);
    }
}
```

```
public class My_Calculation extends
Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the
given numbers:"+z);
    }

    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calculation demo = new
My_Calculation();
        demo.addition(a, b);
        demo.Subtraction(a, b);
        demo.multiplication(a, b);
    }
}
```

# Inheritance

- In the given program, when an object to My\_Calculation class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



# Inheritance

- Note – A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

# Inheritance - Super

- The `super` keyword is similar to `this` keyword. Following are the scenarios where the `super` keyword is used.
  - It is used to differentiate the members of superclass from the members of subclass, if they have same names.
  - It is used to invoke the superclass constructor from subclass.



# Inheritance - Super

- Differentiating the Members

- If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.
  - `super.variable`
  - `super.method( )`;

# Inheritance

- 13 – oo/inheritance/Sub\_class.java

# Inheritance - Invoking Superclass Constructor

- If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.
- `super(values);`

# Inheritance - Invoking Superclass Constructor

```
class Superclass {
    int age;

    Superclass(int age) {
        this.age = age;
    }

    public void getAge() {
        System.out.println("The
value of the variable named age
in super class is: " +age);
    }
}
```

```
public class Subclass extends
Superclass {
    Subclass(int age) {
        super(age);
    }

    public static void
main(String args[]) {
        Subclass s = new
Subclass(24);
        s.getAge();
    }
}
```

# Inheritance - IS-A Relationship

- IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class Animal {  
}
```

```
public class Mammal extends Animal {  
}
```

```
public class Reptile extends Animal {  
}
```

```
public class Dog extends Mammal {  
}
```

# Inheritance - IS-A Relationship

- Now, based on the above example, in Object-Oriented terms, the following are true
- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.
- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

# Inheritance - IS-A Relationship

- With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.
- We can assure that Mammal is actually an Animal with the use of the instance operator.

# Inheritance - IS-A Relationship

```
class Animal {  
}  
  
class Mammal extends Animal {  
}  
  
class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal); // true  
        System.out.println(d instanceof Mammal); // true  
        System.out.println(d instanceof Animal); // true  
    }  
}
```



# Inheritance - Interface

- Since we have a good understanding of the `extends` keyword, let us look into how the `implements` keyword is used to get the IS-A relationship.
- Generally, the `implements` keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

```
public interface Animal {  
}
```

```
public class Mammal implements Animal {  
}
```

```
public class Dog extends Mammal {  
}
```

# Inheritance - instanceof

```
interface Animal{}  
class Mammal implements Animal {}  
  
public class Dog extends Mammal {  
  
    public static void main(String args[]) {  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal); // true  
        System.out.println(d instanceof Mammal); // true  
        System.out.println(d instanceof Animal); // true  
    }  
}
```

# Inheritance - HAS-A relationship

- These relationships are mainly based on the usage. This determines whether a certain class HAS-A certain thing. This relationship helps to reduce duplication of code as well as bugs.

```
public class Vehicle {}  
public class Speed {}
```

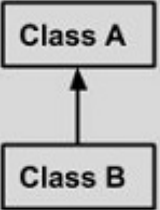
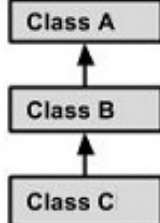
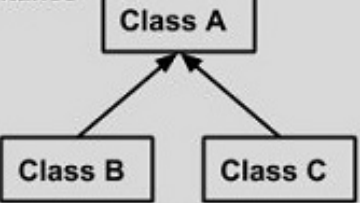
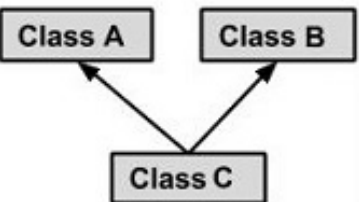
```
public class Van extends Vehicle {  
    private Speed sp;  
}
```

# Inheritance - HAS-A relationship

- This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.
- In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

# Types of Inheritance

- A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class.

<b>Single Inheritance</b>  <pre>graph BT; B[Class B] --&gt; A[Class A]</pre>	<pre>public class A {     ..... } public class B extends A {     ..... }</pre>
<b>Multi Level Inheritance</b>  <pre>graph BT; C[Class C] --&gt; B[Class B]; B --&gt; A[Class A]</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends B {.....}</pre>
<b>Hierarchical Inheritance</b>  <pre>graph BT; B[Class B] --&gt; A[Class A]; C[Class C] --&gt; A</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends A {.....}</pre>
<b>Multiple Inheritance</b>  <pre>graph BT; C[Class C] --&gt; A[Class A]; C --&gt; B[Class B]</pre>	<pre>public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance</pre>

# Uppgift

- Kolla in uppgiften Dogs i repot:
- `exercises/03 - inheritance_dogs/description.txt`

# Overriding

- In the previous section, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.
- The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.
- In object-oriented terms, overriding means to override the functionality of an existing method.

# Overriding

```
class Animal {  
    public void move() {  
        System.out.println("Animals can move");  
    }  
}  
  
class Dog extends Animal {  
    public void move() {  
        System.out.println("Dogs can walk and run");  
    }  
}
```

```
public class TestDog {  
  
    public static void main(String args[]) {  
        Animal a = new Animal(); // Animal reference  
        and object  
        Animal b = new Dog();    // Animal reference  
        but Dog object  
  
        a.move(); // runs the method in Animal class  
        b.move(); // runs the method in Dog class  
    }  
}
```

Output:  
Animals can move  
Dogs can walk and run



# Overriding

- In the above example, you can see that even though b is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.
- Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

# Overriding

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
    public void bark() {
        System.out.println("Dogs can bark");
    }
}
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

```
public class TestDog {

    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and
        object
        Animal b = new Dog();       // Animal reference but Dog
        object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
        b.bark();

    }
}
```

Output:  
TestDog.java:26: error: cannot find symbol

```
        b.bark();
            ^
```

symbol: method bark()

location: variable b of type Animal

1 error

# Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

# Using the super keyword

- When invoking a superclass version of an overridden method the super keyword is used.

# Overriding

```
class Animal {  
    public void move() {  
        System.out.println("Animals can move");  
    }  
}  
  
class Dog extends Animal {  
    public void move() {  
        super.move(); // invokes the super class method  
        System.out.println("Dogs can walk and run");  
    }  
}
```

```
public class TestDog {  
  
    public static void main(String args[]) {  
        Animal b = new Dog(); // Animal reference but Dog object  
        b.move(); // runs the method in Dog class  
    }  
}
```

Output:  
Animals can move  
Dogs can walk and run

# Uppgift

- Kolla in uppgiften Override i repot:
- `exercises/05 - override/description.txt`

# Polymorphism

- Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.
- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

# Polymorphism

- It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
- The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
- A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.



# Polymorphism

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

- Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples
  - A Deer IS-A Animal
  - A Deer IS-A Vegetarian
  - A Deer IS-A Deer
  - A Deer IS-A Object

# Polymorphism

- When we apply the reference variable facts to a Deer object reference, the following declarations are legal

```
Deer d = new Deer();
```

```
Animal a = d;
```

```
Vegetarian v = d;
```

```
Object o = d;
```

- All the reference variables d, a, v, o refer to the same Deer object in the heap.

# Virtual Methods

- In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.
- We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

# Polymorphism

- Study code/13 - oo/polymorphism/Employee.java
- Now suppose we extend Employee class as in code/13 - oo/polymorphism/Salary.java
- Now, you study the following program carefully and try to determine its output: code/13 - oo/polymorphism/VirtualDemo.java
- Output
  - Constructing an Employee
  - Constructing an Employee
  - Call mailCheck using Salary reference --
  - Within mailCheck of Salary class
  - Mailing check to Mohd Mohtashim with salary 3600.0
  - Call mailCheck using Employee reference--
  - Within mailCheck of Salary class
  - Mailing check to John Adams with salary 2400.0

# Polymorphism

- Here, we instantiate two Salary objects. One using a Salary reference s, and the other using an Employee reference e.
- While invoking s.mailCheck(), the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

# Polymorphism

- `mailCheck()` on `e` is quite different because `e` is an `Employee` reference. When the compiler sees `e.mailCheck()`, the compiler sees the `mailCheck()` method in the `Employee` class.
- Here, at compile time, the compiler used `mailCheck()` in `Employee` to validate this statement. At run time, however, the JVM invokes `mailCheck()` in the `Salary` class.
- This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

# Uppgift

- Kolla in uppgiften Polymorphism i repot:
- `exercises/06 - polymorphism/description.txt`

# Abstraction

- As per dictionary, abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.
- Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.
- In Java, abstraction is achieved using Abstract classes and interfaces.



# Abstract Class

- A class which contains the abstract keyword in its declaration is known as abstract class.
  - Abstract classes may or may not contain abstract methods, i.e., methods without body ( `public void get();` )
  - But, if a class has at least one abstract method, then the class must be declared abstract.
  - If a class is declared abstract, it cannot be instantiated.
  - To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
  - If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

# Abstract example

- 13 - oo/abstract/Employee.java
- You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.
- Now you can try to instantiate the Employee class with code/13 - oo/abstract/AbstractDemo.java
- Output:
  - Employee.java:46: Employee is abstract; cannot be instantiated
  - Employee e = new Employee("George W.", "Houston, TX", 43);
  - ^
  - 1 error

# Inheriting the Abstract Class

- We can inherit the properties of Employee class just like concrete class here: `code/13 - oo/abstract/Salary.java`
- Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

# Abstract Methods

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.
  - `abstract` keyword is used to declare the method as abstract.
  - You have to place the `abstract` keyword before the method name in the method declaration.
  - An abstract method contains a method signature, but no method body.
  - Instead of curly braces, an abstract method will have a semi colon (;) at the end.
  - `code/13 - oo/abstract2/Employee.java`

# Abstract Methods

- Declaring a method as abstract has two consequences
  - The class containing it must be declared as abstract.
  - Any class inheriting the current class must either override the abstract method or declare itself as abstract.
- Note – Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

# Abstract Methods

- Suppose Salary class inherits the Employee class, then it should implement the computePay() method as shown below
- `code/13 - oo/abstract2/Salary.java`

# Uppgift

- `exercises/07 - abstract_class/description.txt`

# Encapsulation

- Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.



# Encapsulation

- To achieve encapsulation in Java
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.

# Encapsulation

- `code/13 - oo/encapsulation/EncapTest.java`
- The public `setXXX()` and `getXXX()` methods are the access points of the instance variables of the `EncapTest` class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.
- `code/13 - oo/encapsulation/RunEncap.java`
- Output:  
Name : James Age : 20

# Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

# Uppgift - Encapsulation

- `exercises/08 - encapsulation/description.txt`

# Interfaces

- An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- 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 implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

# Interfaces – similarities to classes

- 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 byte code 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.

# Interfaces – differences to classes

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

# Declaring Interfaces

- Interfaces have the following properties
  - An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - Methods in an interface are implicitly public.

```
interface Animal {  
    public void eat();  
    public void travel();  
}
```



# Implementing Interfaces

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the `implements` keyword to implement an interface. The `implements` keyword appears in the class declaration following the `extends` portion of the declaration.

# Implementing Interfaces

- `code/13 - oo/interface/MammalInt.java`
- Output
  - `Mammal eats`
  - `Mammal travels`

# Implementing Interfaces

- When overriding methods defined in interfaces, there are several rules to be followed
  - Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
  - The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
  - An implementation class itself can be abstract and if so, interface methods need not be implemented.

# Implementing Interfaces

- When implementing interfaces, there are several rules
  - A class can implement more than one interface at a time.
  - A class can extend only one class, but implement many interfaces.
  - An interface can extend another interface, in a similar way as a class can extend another class.

# Extending Interfaces

- An interface can extend another interface in the same way that a class can extend another class. The `extends` keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
- `code/13 - oo/extending_interface/Sports.java`
- The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

# Extending Multiple Interfaces

- A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The `extends` keyword is used once, and the parent interfaces are declared in a comma-separated list.
- For example, if the `Hockey` interface extended both `Sports` and `Event`, it would be declared as

```
public interface Hockey extends Sports, Event
```

# Tagging Interfaces

- The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the `MouseListener` interface in the `java.awt.event` package extended `java.util.EventListener`, which is defined as

```
package java.util;  
public interface EventListener  
{}
```

# Tagging Interfaces

- An interface with no methods in it is referred to as a tagging interface. There are two basic design purposes of tagging interfaces –
  - **Creates a common parent** – As with the `EventListener` interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends `EventListener`, the JVM knows that this particular interface is going to be used in an event delegation scenario.
  - **Adds a data type to a class** – This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.



# Uppgift - Interface

- `exercises/09 - interfaces/description.txt`