University American College Skopje

School of Computer Science and Information Technologies

Course: Programming Languages

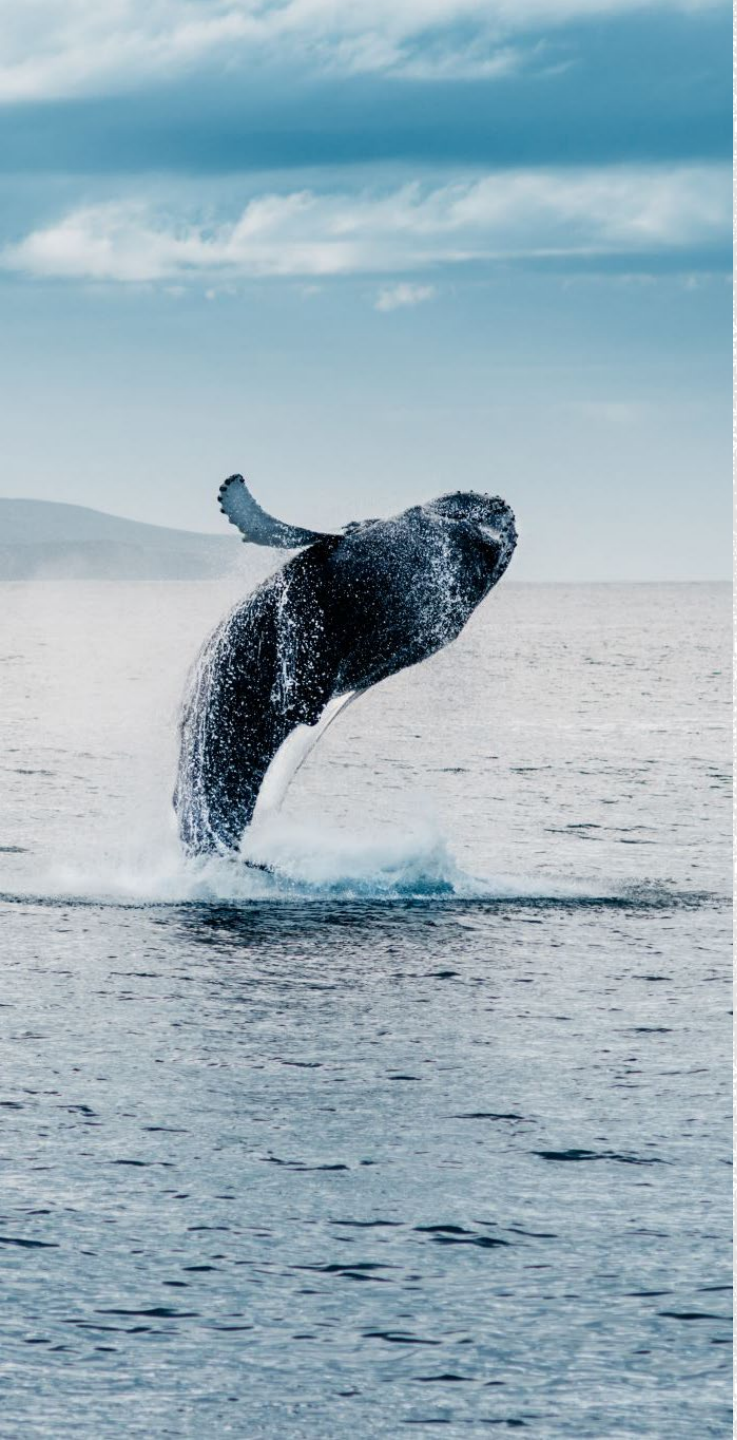
Polymorphism. Interfaces

Lecture 7

POLYMORPHISM

Polymorphism = ability of the same method to cause different behaviors depending on the object invoking it

In other words, the definition of the method depends on the class in which it had been defined

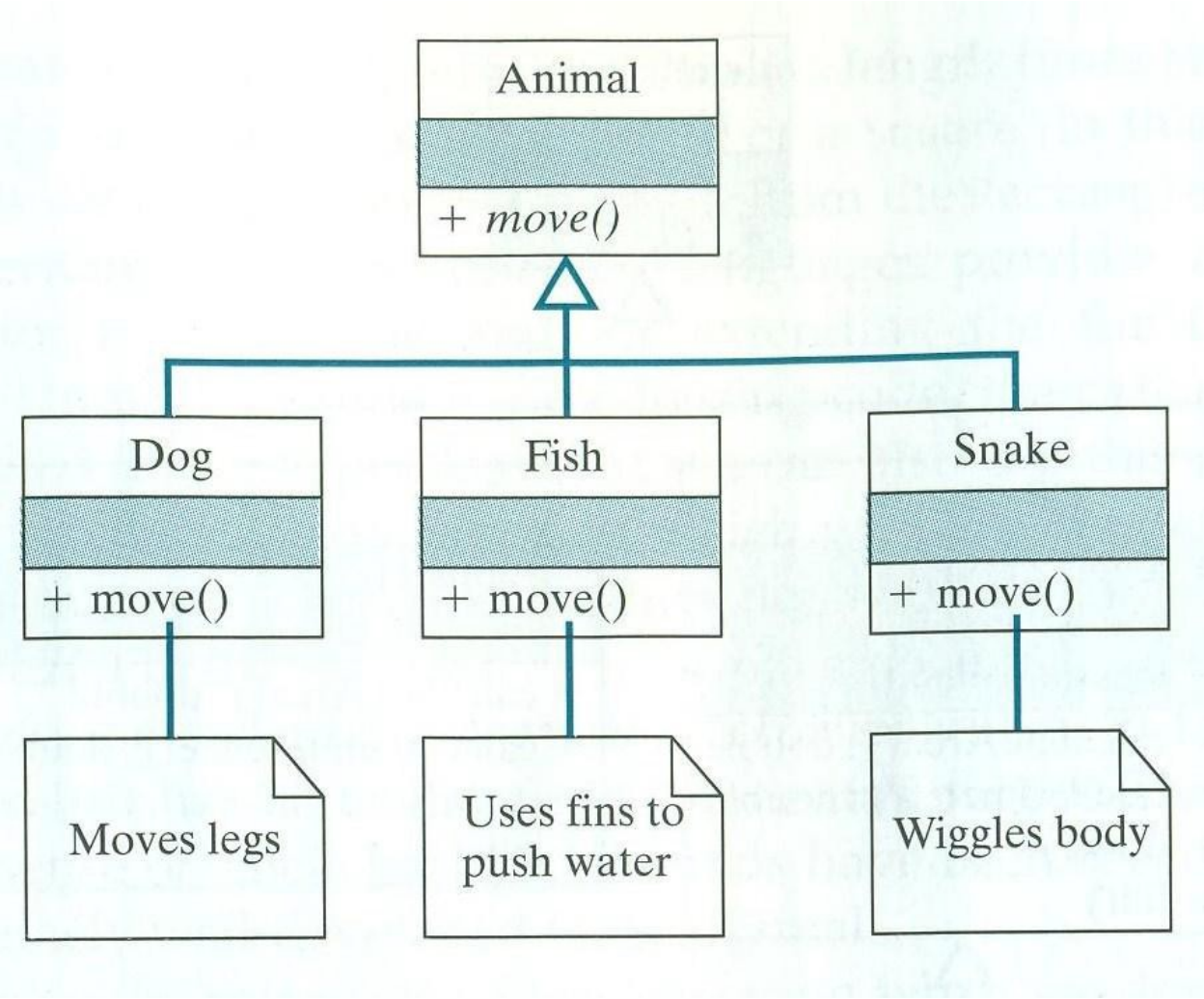


POLYMORPHISM

- Example: How animals move
  - Dog – moves its body using its legs
  - Fish – pushes water away with its fins
  - Snake – slithers on the ground by twisting its entire body

- In other words, every *definition* of the behavior of moving is different for every animal – yet they all *move*

POLYMORPHISM



POLYMORPHISM AND OVERLOADING

- The difference between these two concepts is that polymorphism performs *overriding* of the function, instead of overloading
- In other words, overriding *replaces* the method's functionality, whereas overloading *expands* it

POLYMORPHISM AND OVERLOADING

If two (or more) methods are	Overridden	Overloaded
Their return values are	Identical	Identical / Different
Their names are	Identical	Identical
Their parameter lists are	Identical	Different
Inheritance is (*)	Necessary	Optional

(*) Overriding may occur only if a derived class redefines a method which is already defined in a base class

Overloading may occur in the same class, if the method is defined to perform actions for different parameter lists, and it is also possible to overload a method of the base class without overriding (i.e. redefining) it

POLYMORPHISM AND OVERRIDING

- Polymorphism is possible because of the principle of object oriented programming, which states that a variable of a superclass may accept a value of the type of the subclass
  - – Since every type of the subclass "is-a" type of the superclass
- In that case, if a method in the superclass is overridden in the subclass(es), which definition will be used will be decided at run time

EXAMPLE

- class Base
- {
  - public Base(String name)
  - {
    - this.name = name;
  - }
  - public void print()
  - {
    - System.out.println("Method of the superclass" + name);
  - }
  - protected String name;
- }

```
class Derived extends Base
{
        public Derived(String name)
        {
                super(name);
        }

        public void print()
        {

        System.out.println("Method
of the subclass " + name);
        }

        public String getName()
        {
                return name;
        }
}
```

EXAMPLE

- In the example, `Derived` is a subclass of `Base`
- `Base` has a method `print()`, which is overridden (redefined) in `Derived`
- `Base` has a protected member, which is inherited in `Derived`
- The constructor of `Derived` invokes the constructor of `Base`

EXAMPLE

```
public class Poly
{
   public static void main(String args[])
   {
       Base b = new Base("b");
       b.print();

       Derived d = new Derived("d");
       d.print();
   }
}
```

Output:

```
Method of the superclass b
Method of the subclass d
```

EXAMPLE

- However, it is possible that a reference of type `Base` obtain a value of an object of type `Derived`
  - A reference to the superclass may contain a value of the subclass, since the subclass “is-a” superclass
    - The reverse does not hold
- In that case the code of the *subclass* is used upon the method invocation

EXAMPLE

…

```
    b = new Derived("b");
    b.print();
```

…

Output:

```
Method of the subclass b
```

OVERRIDING IS IRREVERSIBLE!

- When the reference of the superclass obtains a value of an object of the subclass, the definitions in the methods of the subclass override the methods of the superclass
- *It is not possible anymore to invoke the method definitions of the superclass, after the reference is given the method definition of the subclass*
  - – Upcasting doesn't help – the overriding cannot be undone

EXAMPLE

…

```
((Base)b).print();
```

…

Изнес:

```
Method of the subclass b
```

- Upcasting `b` won't invoke the definition of the `print()` method of the superclass – the object had been *created* using the methods of the `Derived` class upon its generation (i.e. at the line `b = new Derived("b");`)

DOWNCASTING

- A reference of the superclass can obtain method definitions of the subclass, if they override the methods defined in the superclass
- In other words, a reference of the superclass may invoke only the methods that are invoked in the superclass, even if their definitions (i.e. code) can be obtained by overriding the method definitions of the subclass

DOWNCASTING

- If the subclass reference obtains a value of the type of the superclass upon instantiation, the methods defined in the subclass which are not defined in the superclass cannot be directly accessed

- For example, the line System.out.println("The name is " + b.getName());
- would cause an error upon compilation, since getName() is a method if the subclass (Derived), but not of the superclass (Base) and b is a reference to the superclass (i.e. Base) and contains only methods of the superclass, even if they had been redefined (i.e. overridden) by the subclass (upon instantiation, i.e. at the line b = new Derived("b");)

DOWNCASTING

- Stilll, since the reference of the superclass is instantiated at a value of the subclass, the reference to the superclass can be downcast to the type of the subclass and thus access all methods of the subclass
- Example:

…

```
System.out.println("The name is " +
  ((Derived)b).getName());
```

…

Would output

```
The name is b
```

DYNAMIC (LATE) BINDING

- If the reference to the superclass obtains a value of the superclass, it cannot be cast into a type of a subclass, not even using explicit casting
- For example:

```
b = new Base("b");
((Derived)b).print();
```

DYNAMIC (LATE) BINDING

- The lines of code in the previous slide will cause a *run-time error*, i.e. the error would be noticed only during the execution of the program
- This is a concept of *dynamic binding* or *late binding*, i.e. linking of the meaning (code) of the method with its invocation during execution
  - In other words, the code of the method is not known during compilation time – this is the essence of polymorphism
  - The linking of the method code with its invocation is called *static* (or *early*) *binding* – this is the case with overloading

Abstract Classes and Methods

- Upon polymorphism, values of types of subclasses are given to references of superclasses, so it makes sense that the methods that will be overridden be declared as *abstract*
- In that case, the subclass *must* override the abstract methods, so that full functionality and complete implementation is ensured
  - And so the class be allowed to instantiate objects
  - Abstract classes and methods offer *incomplete implementation* and that is the reason that instantiating objects out of abstract classes is not allowed

EXAMPLE

```
abstract class A
{
    A()
    {
          System.out.println("Default
    constructor of A");
    }

    public abstract void m();
}
```

```
class B extends A
{
          B()
          {
                    super();
          }

          public void m()
          {

          System.out.println("Overriding
of the abstract method m() inherited by
class A");
          }
}
```

```
public class Abstracts
{
          public static void main(String[] args)
          {
                    A a;
                    a = new B();
                    a.m();
          }
}
```

The method `m()` is *declared* in the class `A` and therefore can be used in the *implementation* by the class `B`

Output:

```
Default constructor of A
Overriding of the abstract method m() inherited by class A
```

Interfaces as data types

- Interfaces have been introduced in Java as a way of simulating multiple inheritance
  - By default, Java does not allow multiple inheritance
- Interfaces are data types, in a sense that they are declared and defined (i.e. have a line for declaration and a body)
  - They are declared using the keyword `interface`

```
interface interfaceName
{…}
```

INTERFACES AND ABSTRACT CLASSES

- Interfaces are the closest to the abstract classes, in a sense that
  - Their methods are all abstract
  - They cannot be instantiated
- But, unlike abstract classes, interfaces
  - Have no constructors
    - It is an interface, not a class
  - Have no non-abstract methods
    - All methods are abstract, in the sense that they don't have definition codes (the keyword `abstract` is default and need not be specified)
  - Contain no variables
    - Constants are allowed, i.e. all variables declared in an interface are implicitly `static` and `final`
  - Have all members as `public` by default

Implementation of Interfaces

- A class may *implement* an interface
- This means that the class must offer definitions for all methods that are declared in the interface
  - If that is not the case, the class must be declared as abstract
- Interfaces are implemented using the keyword `implements`

IMPLEMENTATION OF INTERFACES

- More than one interface may be implemented
  - This is a simulation of multiple inheritance
- Example:

```
class B extends A implements interface1,
  interface2, …
```

  In this case, class `B` must implement all methods declared in `interface1` and `interface2`

- Every interface can also extend (i.e. inherit from) other interfaces, but it cannot implement another interface
- Example:

```
interface i3 extends i1, i2, …
```

  Now every class that implements `i3` must implements methods from `i3`, `i1` and `i2`

USING INTERFACES AS DATA TYPES

- It is legal to declare a reference to an interface
- Still, the reference cannot be instantiated – the reference to the interface must obtain a value of a type that implements that interface
- Afterwards, that reference may access only methods declared in the interface, using the definitions of the class through which it was instantiated

EXAMPLE

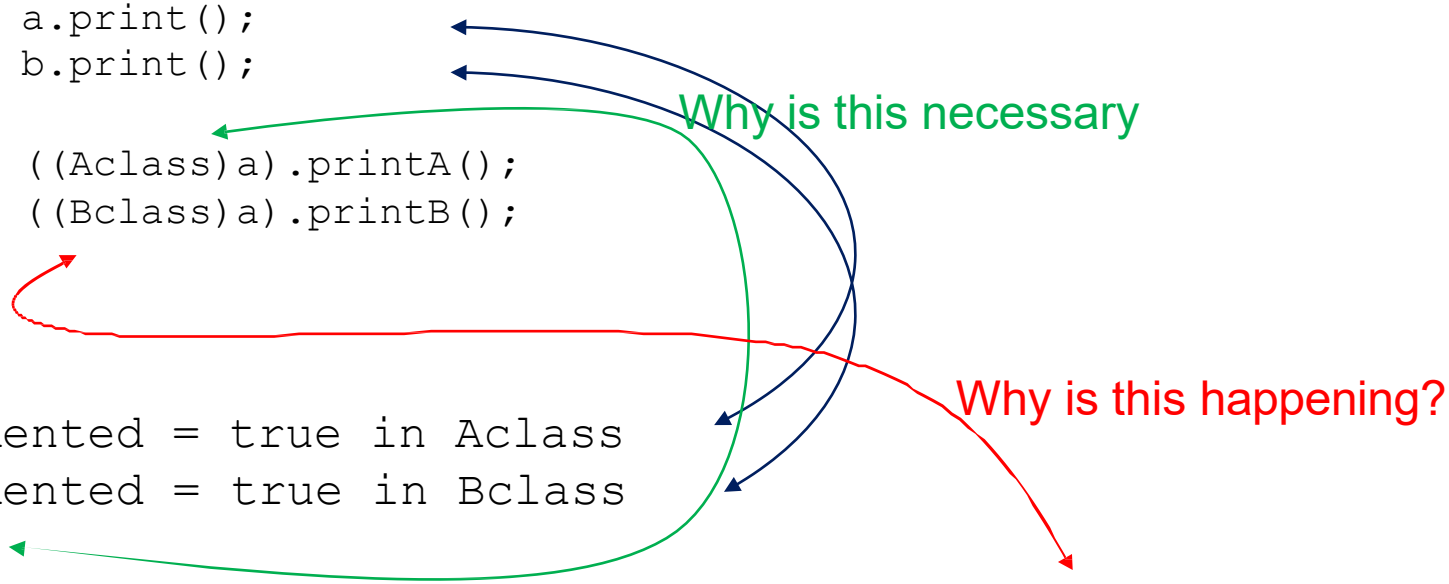
```
class Aclass implements Implementable
{
    public void print()
    {
          System.out.println("Implemented = " + implemented + " in Aclass");
    }

    public void printA()
    {
          System.out.println("Aclass");
    }
}
```

```
interface Implementable
{
          boolean implemented = true;

          void print();
}
```

```
class Bclass implements Implementable
{
    public void print()
    {
          System.out.println("Implemented = " + implemented + " во Bclass");
    }

    public void printB()
    {
          System.out.println("Bclass");
    }
}
```

EXAMPLE

```
public class Interfaces
{
    public static void main(String[] args)
    {
        Implementable a = new Aclass();
        Implementable b = new Bclass();

        a.print();
        b.print();

        ((Aclass)a).printA();
        ((Bclass)a).printB();
    }
}
```



Output:

```
Implemented = true in Aclass
Implemented = true in Bclass
Aclass
Exception in thread "main" java.lang.ClassCastException:
Aclass cannot be cast to Bclass
        at Interfaces.main(Interfaces.java:12)
```

Using Interfaces as Method Arguments

- In the same way as interface types may be used inside of a method, they may also be specified as method arguments
- In that case, *any* class instance that implements that interface may be given as an argument to the method which has the type of the interface as an argument
  - This way, Java simulates multiple inheritance

EXAMPLE

```
public class Interfaces
{
    static void m(Implementable i)
    {
         i.print();
    }

    public static void main(String[] args)
    {
         …
         m(a);
         m(b);
    }

}
```

Output:

```
…
Implemented = true in Aclass
Implemented = true in Bclass
```

QUESTIONS