Java Programming Module Polymorphism and Interfaces

DHBW Duale Hochschule Baden-Württemberg Mannheim

Prof. Dr. Holger D. Hofmann

Abstract Classes

- Abstract classes contain at least one abstract member

```
001 abstract class clsTestAbstract {
002  …
005  public abstract void doSomething();// no method body
006  …
015 }
```

- An abstract class cannot be instantiated
- A derived class can be instantiated if all abstract members have been implemented

- … are used if a method‘s implementation is subclass-specific and cannot be reused by the base class

Example 1: Abstract classes

```
abstract class clsAbstract {}
```

Does `clsAbstract a = new clsAbstract();` work?

Cannot instantiate the type clsAbstract

public class clsTest extends clsAbstract {}

Does `clsTest t = new clsTest();` work?

Example 2: Abstract classes

```
public abstract class clsAbstract2 {
        public abstract int getNumber();
}

public class clsTest2 extends clsAbstract2 {
        @Override
        public int getNumber() {
                // TODO Auto-generated method stub
                return 0;
        }
}

clsTest2 t2 = new clsTest2(); // works!
```

Method overriding

Method Overriding

- Inherited methods can be overridden

```
01        class Human{
02           //Overridden method
03           public void eat()
04           {
05                    System.out.println("Human is eating");
06           }
07        }
08        class Boy extends Human{
09           //Overriding method
10           public void eat(){
11                    System.out.println("Boy is eating");
12           }
13           public static void main( String args[]) {
14                    Boy obj = new Boy();
15                    //This will call the child class version of eat()
16                    obj.eat();
17           }
18        }
```

Source: beginnersbook.com

Method Overloading

- Methods and operators are treated differently in Java
- Methods can be overloaded, operators not
- For example, the String class redefines the + operator, but you cannot do that for one of your own classes
- The Number class provides compareto()-Methods rather then redifining the == operator

Example: Operator overloading in C++

```
01      #include<iostream>
02      using namespace std;
03
04      class Complex {
05      private:
06              int real, imag;
07      public:
08              Complex(int r = 0, int i =0)  {real = r;   imag = i;}
09
10              Complex operator + (Complex const &obj) {
11                      Complex res;
12                      res.real = real + obj.real;
13                      res.imag = imag + obj.imag;
14                      return res;
15              }
16              void print() { cout << real << " + i" << imag << endl; }
17      };
18
19      int main()
20      {
21              Complex c1(10, 5), c2(2, 4);
22              Complex c3 = c1 + c2; // An example call to "operator+"
23              c3.print();
24      }
```

Source: geeksforgeeks.org

Wrapper Classes

- There exists a wrapper class for each elemental data type
- They support the creation of objects from elemental types
- all classes have two constructor forms:
  - a constructor that takes the primitive type and creates an object, e.g., Character(char), Integer(int)
  - a constructor converting a String into an object, e.g., Integer("12")
- all classes override equals(), hashCode() and toString() in Object
  - equals() returns true if the values of the compared objects are the same
  - hashCode() returns the same hashcode for objects of the same type having the same value
  - toString() returns the string representation of the objects value

Wrapper Class	Elemental Type
Byte	byte
Short	short
Integer	int
Long	long
Double	double
Float	float
Boolean	boolean
Character	char
Void	void

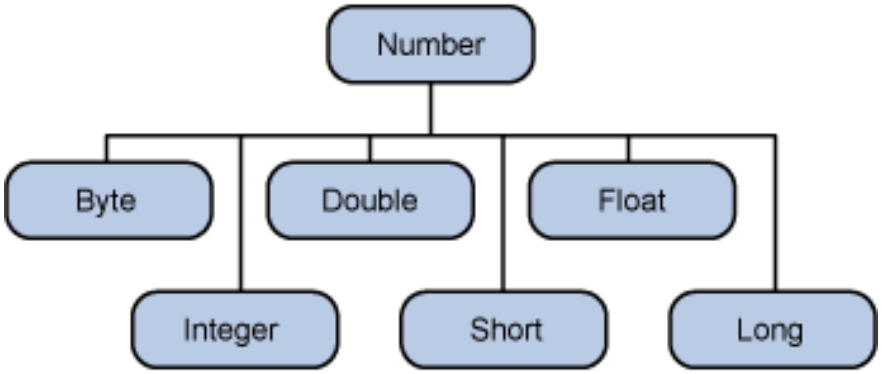
Wrapper Classes

- Creation:

```
public Float(float f)
public Float(double d)
public Float(String s)  throws NumberFormatException
```

- Getting the value (numeric data types):

```
public boolean booleanValue()
public char charValue()
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```



- Getting a string representation

```
public String toString()
```

- String parsing

```
public static int parseInt(String s)
```

Methods implemented by Number Class

Method	Description

`byte byteValue()`
`short shortValue()`
`int intValue()`
`long longValue()`
`float floatValue()`
`double doubleValue()`
	Converts the value of this `Number` object to the primitive data type returned.

`int compareTo(Byte anotherByte)`
`int compareTo(Double anotherDouble)`
`int compareTo(Float anotherFloat)`
`int compareTo(Integer anotherInteger)`
`int compareTo(Long anotherLong)`
`int compareTo(Short anotherShort)`
	Compares this `Number` object to the argument.

`boolean equals(Object obj)`	Determines whether this number object is equal to the argument. The methods return `true` if the argument is not `null` and is an object of the same type and with the same numeric value. There are some extra requirements for `Double` and `Float` objects that are described in the Java API documentation.

[Source: Sun Java Tutorial]

Wrapper Classes

- Implicit conversion (casting) between Wrapper Classes and elemental data types is called Autoboxing (and Autounboxing)
- Example:

```
public class Test
{
   public static void withAutoboxing(int arg)
      {
        Integer i = arg; //Wrapper Object
        int j = i + 1;   //Elemental type and Wrapper Object
        System.out.println(i + " " + j);
      }

      public static void main(String[] args)
      {
        withAutoboxing(new Integer(17));
      }
}
```

Wrapper Classes

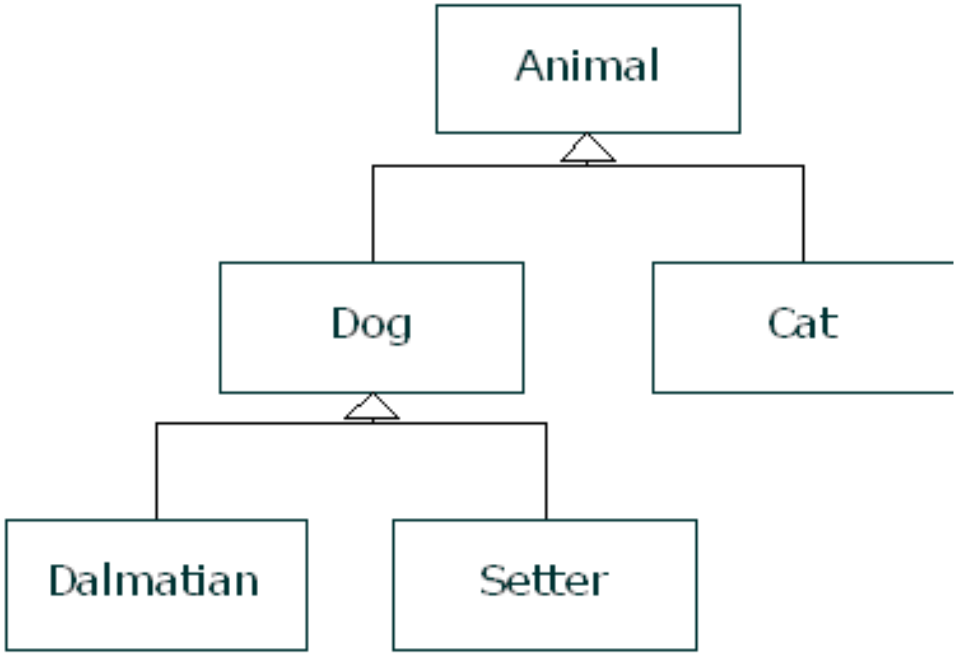
- Wrapper Classes are objects and thus are represented by references
- However, changing their contents in, e.g., a method does not work

```
public  static void doAdd(Integer i)
 {
 ++i;
 return;
 }
```

- All predefined Wrapper Classes are immutable!

Object-oriented Concepts: Polymorphism

- Result of inheritance
- Polymorphism: Objects or derived classes can be used as objects of base classes
- Example: Animal Inheritance



- Class Dog offers method "bark". Both objects of classes "Dalmatian" and class "Setter" can use the bark method.

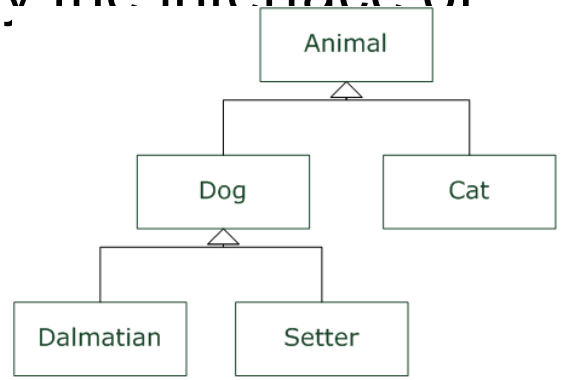
Polymorphism Example

- `public class Student extends Person`
  - -> Student Objects can be used as Person Objects
- Example:

```
1  Student s = new Student();
2  s.m_name = "Paul";
3  s.m_age = 21;
4  s.getOlder(); //method from base class
5
6  Person p = s;
```

Polymporphism: Liskov Substitution Principle

- Liskov Substitution Principle
  Subclasses should be accessible by the interface of their base class
  - Subclasses are not to "hide" parts of their signature or modify it
    This does not work in Java;
    only possible with "bad programming" (see below)




```
// Java Example
public class clsFather {
private int m_age;
public int getAge() { return m_age; }
}


public class clsSon extends clsFather {
public int getAge() { throw new RuntimeException("nothing"); }
}
```

The Object Class

- Every class in Java in derived from the Object Class
- The Object class provides elemental methods:
  - boolean equals(Object obj) – object comparison
  - protected Object clone() – object copying
  - String toString() – string representation of object
  - int hashCode() – creates numeric key for object identification
- Elemental object methods may be overwritten on demand

Interfaces

- Interfaces augment the signature of classes and contain
  - Abstract Methods
  - Constants
- They do not contain any implementation (Java < Version 8)

```
001 public interface ISize
002 {
003   public int length();
004   public int height();
005   public int width();
006 }
```

```
class XYZ implements ISize { //…
```

Example: Interfaces

```
1          package interfaces;
2
3          public interface ISize {
4                    public int length();
5                    public int height();
6                    public int width();
7          }
```

```
01         package interfaces;
02
03         public class clsApple implements ISize{
04                   @Override
05                   public int length() { return 2; }
06
07                   @Override
08                   public int height() { return 3; }
09
10                   @Override
11                   public int width() {  return 4; }
12         }
```

```
01         package interfaces;
02
03         public class clsPear implements ISize{
04                   @Override
05                   public int length() { return 2; }
06
07                   @Override
08                   public int height() { return 3; }
09
10                   @Override
11                   public int width() {  return 4; }
12         }
```

Polymorphism and Interfaces

```
package polymorphism;

public class clsPerson implements ICallable{
public void call(String phonenumber) {
//accept call, etc.
}
}
```

```
package polymorphism;

public class clsIOTDevice implements
ICallable{
public void call(String phonenumber) {
//accept call, etc.
}
}
```

```
package polymorphism;

public class clsStudent extends
clsPerson {
//add own methods
}
```

```
package polymorphism;

public interface ICallable {
public void call(String phonenumber);
}
```

Polymorphism and Interfaces

```
package polymorphism;

public class clsMain {

public static void main(String[] args) {
     clsPerson p = new clsPerson();
    clsStudent s = new clsStudent();

    clsPerson refPerson;
    clsStudent refStudent;
    ICallable refInterface;

    refPerson = p;
    refPerson = s; //a student is also a person

    refStudent = s;
    refStudent = p; //does not work!!!

    refInterface = p;//polymorphism via interface!
    refInterface = s;//polymorphism via interface!
}
}
```