5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

in the past:

- Programming languages have a fixed set of data types with predefined operations.

today:

- Since new application domains are identified all the time, it must be possible to add new data types and the corresponding operations.
- In addition the engineering of big systems must be better supported.

Class

- blueprint/template for the construction of new objects.

Object

- Representation of an element of a specific class with the operations that can be executed on it

Classes and objects support:

- **Abstraction**
  provide an interface & hide the actual implementation.
- **Data encapsulation**
  access to the data only through predefined methods.
- **Complex structures**
  construction of complex data structures and their operators.
- **Reusability**
  provide all-purpose program components.

An object has

- a **state** (defined by the values of its variables)
- a **behavior** (defined by its methods)

Example:

- A coin shows on its upside either “heads” or “tails”
- The upside can be changed, for example by a “coin toss”.
- The “state” of the object *coin* is its current upside (heads or tails).
- The “behavior” of the coin is that the coin can be tossed.
- The “behavior” of the coin can change its state.

A class is

- a blueprint for an object,
  objects are generated using it as a template.

Example: **`String`**

- The class **`String`** is used to generate **`String`** objects.
- Every **`String`** object contains a sequence of characters defining its state.
- On every **`String`** object a set of methods can be applied.
- These methods offer services (e.g., **`toUppercase()`**, **`equals()`** etc.), i.e., they define the behavior of a **`String`** object.
- Here, the behavior does not change the state of the object, but yields information and values for new objects.

First example, directory `K5B01E_HeadsOrTails`:
`Java` class for “coin tosses”, file `CoinToss.java`

```java
1  public class CoinToss {
2     private final int HEADS = 1;
3     private final int TAILS = 0;
4     private int upside;
5
6     public CoinToss() {toss();}
7
8     public void toss() {upside = (int) (Math.random() * 2);}
9
10    public boolean isHeads() {return (upside == HEADS);}
11
12    @Override public String toString() {
13       String top;
14       if (upside == HEADS) top = "HEADS";
15       else top = "TAILS";
16       return top;
17    }
18 }
```

line 1	class header
lines 2-4	instance variables

lines 6	constructor
lines 8-17	methods for objects

Exampe: application of the ‘coin tosses’, `HeadsOrTails.java`

```java
public class HeadsOrTails {
   public static void main (String[] args) {
      final int NUMBER_THROWS = 1000;
      int heads = 0, tails = 0;

      CoinToss myCoin = new CoinToss();

      for (int count=1; count <= NUMBER_THROWS; count++) {
         myCoin.toss();
         if (myCoin.isHeads())
            heads++;
         else
            tails++;
      }

      System.out.println(
            "In " + NUMBER_THROWS + "␣throws we had "
             + heads + "␣times heads and "
             + tails + "␣times tails.");
   }
}
```

- Compilation, e.g., with
  `javac HeadsOrTails.java`
- Then `HeadsOrTails.java` must have access to the file `CoinToss.class`. If this cannot be found, the compiler tries to generate it using
  `javac CoinToss.java`

  In this case, `CoinToss.java` must be available.
- Simplest solution: all files in the same directory.
- For the assignments you best create a new directory for every task!
- In big projects: use compiler option `-classpath` to define a search path. ($\rightsquigarrow$ `man java`)
- In `moodle` the files of a task are stored together in a directory.

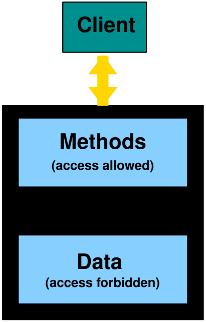
Encapsulation:

- Objects can be seen from two viewpoints: internally and externally.
- From an **internal viewpoint**, an object is a collection of variables and methods accessing these variables.

  (This is not fully exact as methods belonging to objects of a class are stored only once for every class, not with every object)
- From an **external viewpoint**, an object is an encapsulated unit offering some services.

  These services define the interface of the object.
- An object is thus an abstraction that hides details of the implementation from the rest of the system.
- An object can interact with other objects by using their services (i.e., by calling their methods).

- An (encapsulated) object can be seen as a black box.
- The inner details remain hidden to the caller of a method.
- An object should be ‘self-controlled’, i.e., every change of the object’s state (the variables) should be done only using the methods provided by the object.



Constructors:

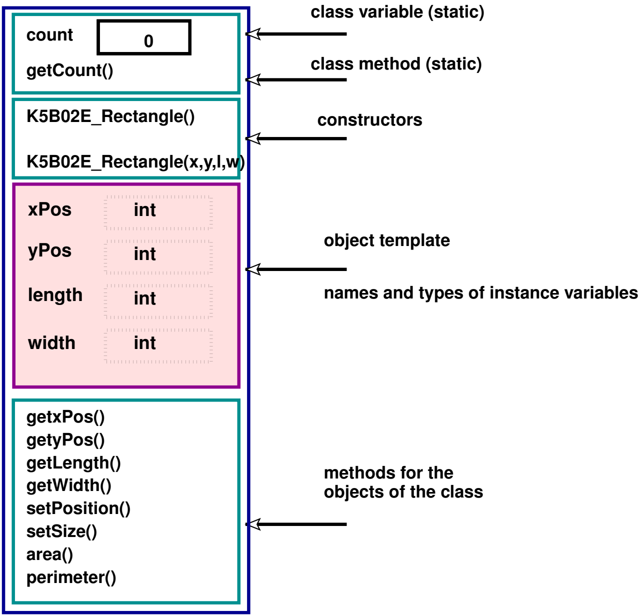
- special methods only used for generating a new object.
- are mostly used to initialize the variables of an object.
- always have the same name as the class.
- can be overloaded.
- can have parameters, but do not return anything.

If not specified, a default constructor is used (corresponding to a constructor with empty parameter list and empty method body).

Example `K5B02E_Rectangle`: rectangles in a plane

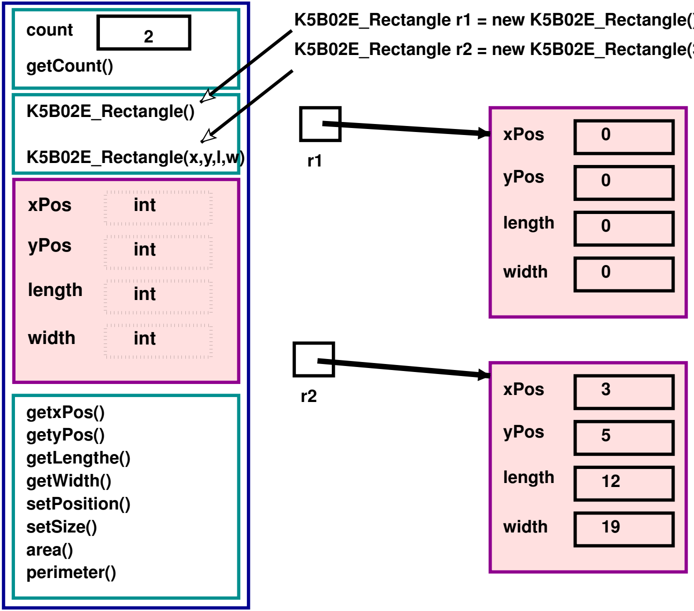
```
public class Rectangle {

  static int count; class variable (static)

  private int xPos, yPos, width, height; instance variables

  Rectangle () { count++; } constructor 1

  Rectangle (int x, int y, int w, int h) { constructor 2
      setPosition (x, y); setSize (w, h); count++; }

  static int getCount() {return count;} class method (static)

  int getxPos ()    { return xPos;} object methods
  int getyPos ()    { return yPos;}
  int getHeight  () { return height;}
  int getWidth () { return width;}

  void setPosition (int x, int y) { xPos = x; yPos = y; }
  void setSize (int w, int h) { width = w; height = h;  }

  int area () { return width * height; }
  int perimeter () { return 2 * ( width + height ); }
}
```

```java
public class RectangleTest {
   public static void main(String[] args) {

      Rectangle r1 = new Rectangle ();
      r1.setSize (4, 12);

      Rectangle r2  = new Rectangle (3, 5, 12, 19);

      System.out.println(
   "Rectangle r1:"
 + "\nx␣␣␣␣␣=␣" + r1.getxPos()   + ",␣y␣␣␣␣␣=␣" + r1.getyPos()
 + "\nWidth= " + r1.getWidth() + ",␣height  = " + r1.getHeight()
 + "\nArea= " + r1.area()    + ",␣perimeter= " + r1.perimeter()
 + "\n\n" +
   "Rectangle r2:"
 + "\nx␣␣␣␣␣=␣" + r2.getxPos()   + ",␣y␣␣␣␣␣=␣" + r2.getyPos()
 + "\nWidth= " + r2.getWidth() + ",␣height  = " + r2.getHeight()
 + "\nArea= " + r2.area()    + ",␣perimeter= " + r2.perimeter()
 + "\n\n"+
   "generated objects: " + Rectangle.getCount ());
     }
}
```



count
0
getCount()
K5B02E_Rectangle()
K5B02E_Rectangle(x,y,l,w)
xPos
int
yPos
int
length
int
width
int
getxPos()
getyPos()
getLength()
getWidth()
setPosition()
setSize()
area()
perimeter()
class variable (static)
class method (static)
constructors
object template
names and types of instance variables
methods for the
objects of the class

```
K5B02E_Rectangle r1 = new K5B02E_Rectangle();
K5B02E_Rectangle r2 = new K5B02E_Rectangle(3,5,12,19);
```



Visibility modifiers

- In Java encapsulation is achieved using visibility modifiers.
- A *modifier* is a reserved word that determines several characteristics (of methods or variables).
  If for example a data element is defined with the modifier **`final`**, this is treated as a constant.
- Java has for options for the visibility of elements of a class (i.e., for methods, variables, constants):
  - **`public`**: can be used from everywhere.
  - **`private`**: can be used only from within the class.
  - **`protected`**: see later (inheritance)
  - *without modifier* (‘package private’): can be used only within the same ‘package’.
- **`public`** variables should be avoided, usually every variable should be ‘private’.

Visibility modifiers for **methods**:

- Methods that offer services included in the interface of the object are usually defined as '`public`'.
- Methods that are only used by other methods of the class ('support methods') should be declared '`private`'.

	`public`	`private`
variables	violation of the encapsulation	support of the encapsulation
methods	service for 'clients'	support for other methods of the class

visibility modifiers for **classes**:

- `public`:
  visible also outside the own 'package'.
- *without modifier* ('package private'):
  visible only in the same 'package'.

`static`: important modifier for **class or object scope** of methods and variables:

- `static`:

  A '`static`' method can be called without the existence of an object (e.g., `Rectangle.getCount()`).

  A '`static`' variable is stored in the memory area of the class (e.g., '`count`' in the 'Rectangle' example).

  With other classes the class name must be used, (e.g., `Rectangle.getCount()` or `Rectangle.count`).
- not `static` (default!):

  Non-`static` variables are stored in the memory area of an object.

  Non-`static` variables and methods thus always refer to a specific object (e.g., `r1.getWidth()`)

`setter` and **`getter`** methods:

- By the principle of encapsulation, a variable should not be directly accessible from the outside.
- Instead:
  - Variables should be declared as **`private`**, in addition optional
  - **`public getter`** method for reading its value and
  - **`public setter`** method for modifying its value

for example:

```
...
private int value;
public int getValue () {
...
}
public void setValue (int newValue) {
...
}
```

Scope of variables:

- The variables and methods defined at class level are also called class members.
- Every class method can access all class members, i.e., all class members are valid in every class method.
- The variables defined at method level are only known in the local method.
- Local variables can hide class variables.

Referencing class members:

- Members are referenced by their name...
- in the *local* object directly by variable or method name
- when hidden by a local variable referenced through `this` in the following form
  `this.Name`
- in a *different* object or a *different* class:
  `ReferenceName.Name` *or even*
  `ReferenceName1.ReferenceName2.Name`

- The keyword **`this`** allows objects to reference themselves
- This can be used to access instance variables hidden by local variables:

```
public class Rectangle {
   static int count;
   private int xPos, yPos, width, height ;
   Rectangle () { count++; }
   Rectangle (int x, int y, int w, int h) {
      setPosition (x, y); setSize (w, h); count++; }

...

   void setSize (int width, int height) {
      this.width = width; this.height = height;   }

   int area () { return width * height; }
   int perimeter () { return 2 * ( width + height ); }
}
```

5 Classes and Objects

In the following:

- class for representing and manipulating rational numbers
- every object of this class represents a rational number
- storage as pair (numerator, denominator).
- with methods for basic arithmetics, test for equality, conversion to Strings

```java
public class RationalNumber {
   private int numerator, denominator;

   //--------------------------------------------------
   //  Constructur:
   //  - Initialization of an object "rational number"
   //  - parameter values assigned to variables
   //  - numerator contains the sign
   //  - representation in canonical form (cancel)
   //  - no check if denominator is 0
   //--------------------------------------------------
   public RationalNumber (int num, int denom) {

      if (denom < 0) {
         num = -num;
         denom = -denom;
      }
      numerator = num;
      denominator = denom;

      cancel();
   }
```

```java
   //-----------------------------------------------
   //  retrieves numerator
   //-----------------------------------------------
   public int getNumerator () {
      return numerator;
   }

   //-----------------------------------------------
   //  retrieves denominator
   //-----------------------------------------------
   public int getDenominator () {
      return denominator;
   }

   //-----------------------------------------------
   // yields the inverse as a rational number
   //-----------------------------------------------
   public RationalNumber inverse () {
      return new RationalNumber (denominator, numerator);
   }
```

```
    //-----------------------------------------------
    //  - addition of two rational numbers
    //  - returns the sum as rational number
    //-----------------------------------------------
    public RationalNumber add (RationalNumber op2) {
       int commonDenominator = denominator * op2.getDenominator();
       int numerator1 = numerator * op2.getDenominator();
       int numerator2 = op2.getNumerator() * denominator;
       int sum = numerator1 + numerator2;
       return new RationalNumber (sum, commonDenominator);
    }

    //-----------------------------------------------
    //  - subtraction (this number - parameter op2)
    //  - returns difference
    //-----------------------------------------------
    public RationalNumber subtract (RationalNumber op2) {
       int commonDenominator = denominator * op2.getDenominator();
       int numerator1 = numerator * op2.getDenominator();
       int numerator2 = op2.getNominator() * denominator;
       int difference = numerator1 - numerator2;
       return new RationalNumber (difference, commonDenominator);
    }
```

```java
    //-----------------------------------------------------
    //  - multiplication of two rational numbers
    //  - returns product as rational number
    //-----------------------------------------------------
    public RationalNumber multiply (RationalNumber op2) {
       int num = numerator * op2.getNumerator();
       int denom = denominator * op2.getDenominator();
       return new RationalNumber (num, denom);
    }

    //-----------------------------------------------------
    //  - division (this number / parameter op2)
    //  - returns quotient as rational number
    //-----------------------------------------------------
    public RationalNumber divide (RationalNumber op2) {
       return multiply (op2.inverse());
    }
```

```java
    //-----------------------------------------------
    //  compare two rational numbers
  // (in canonical form by construction)
    //-----------------------------------------------
    public boolean equals (RationalNumber op2) {
       return ( numerator == op2.getNumerator()
                        && denominator == op2.getDenominator() );
    }

    //-----------------------------------------------
    //  transform a rational number to a String
    //-----------------------------------------------
    @Override public String toString () {
       String result;
       if (numerator == 0)
          result = "0";
       else
          if (denominator == 1)
             result = numerator  + "";
          else
             result = numerator + "/" + denominator;
       return result;
    }
```

```
    //--------------------------------------------------
    //  cancel a rational number
 // (i.e., convert it into its canonical form)
    //--------------------------------------------------
   private void cancel () {
      if (numerator != 0) {
         int common = gcd (Math.abs(numerator), denominator);
         numerator = numerator / common;
         denominator = denominator / common;
      }
    }

    //--------------------------------------------------
    //  - greatest common divisor of two integers
    //  - returns gcd
    //--------------------------------------------------
   private int gcd (int number1, int number2) {
      while (number1 != number2)
         if (number1 > number2)
            number1 = number1 - number2;
         else
            number2 = number2 - number1;
      return number1;
    }
}
```

```java
import java.util.Scanner;
import java.util.*;

public class RationalNumbers {
  public static void main (String[] args) {
    Scanner sc = new Scanner(System.in);

    RationalNumber r1, r2;

    int numerator, denominator;

    char operator;

    System.out.println("Input:\n" +
        "(Syntax: number|number [+-*/] number|number [+-*/]... )" );

    String input = sc.nextLine();

    StringTokenizer tokens =
            new StringTokenizer (input, "| ",true);
```

```
    numerator = Integer.parseInt (tokens.nextToken () );
    tokens.nextToken();
    denominator = Integer.parseInt (tokens.nextToken () );

    r1 = new RationalNumber (numerator, denominator);

    while (tokens.hasMoreTokens () ) {
      operator = tokens.nextToken().charAt(0);

      numerator = Integer.parseInt (tokens.nextToken () );
      tokens.nextToken();
      denominator = Integer.parseInt (tokens.nextToken () );

      r2 = new RationalNumber (numerator, denominator);

      switch (operator) {
        case '+': r1 = r1.add (r2);       break;
        case '-': r1 = r1.subtract (r2);   break;
        case '*': r1 = r1.multiply (r2); break;
        case '/': r1 = r1.divide (r2);     break;
      }
    }
    System.out.println ("result: " + r1.toString() );
  }
}
```