**Computer Science II, D-BAUG** FS 2015
Departement of Computer Science
ETH Zurich

**Assignment 8**
Out: 22th of April 2015
Due: 29th of April 2015

---

# 1 Inheritance applied to finding roots of a function

For this weeks exercise you will implement a small application mainly from a specification, very similar to task 3 of the bonus exercise of last week.
We are not using the judge for this exercise - so mail your TA for feedback to your solution. The goal of this exercise is to implement the Newton method for finding roots of functions, which you should be familiar with. In case you need a refresher we forward you to the according Wikipedia article: Newtons method

## 1.1 Overview over the classes

Below you find a so called UML Diagram of the project. UML is a standard way of drawing the architecture of software and you might encounter it more often in the course of your education. For the sake of this exercise its a nice way of illustrating how we expect you to build the application and should be easy enough to understand. As you can see in the diagram the whole project is supposed to consist of six classes. The only one we give you up front is the class $TestNewton$ which we provide here: `http://lec.inf.ethz.ch/baug/informatik2/2015/ex/ex08/TestNewton.java`.
We describe all classes in more detail in the following sections:

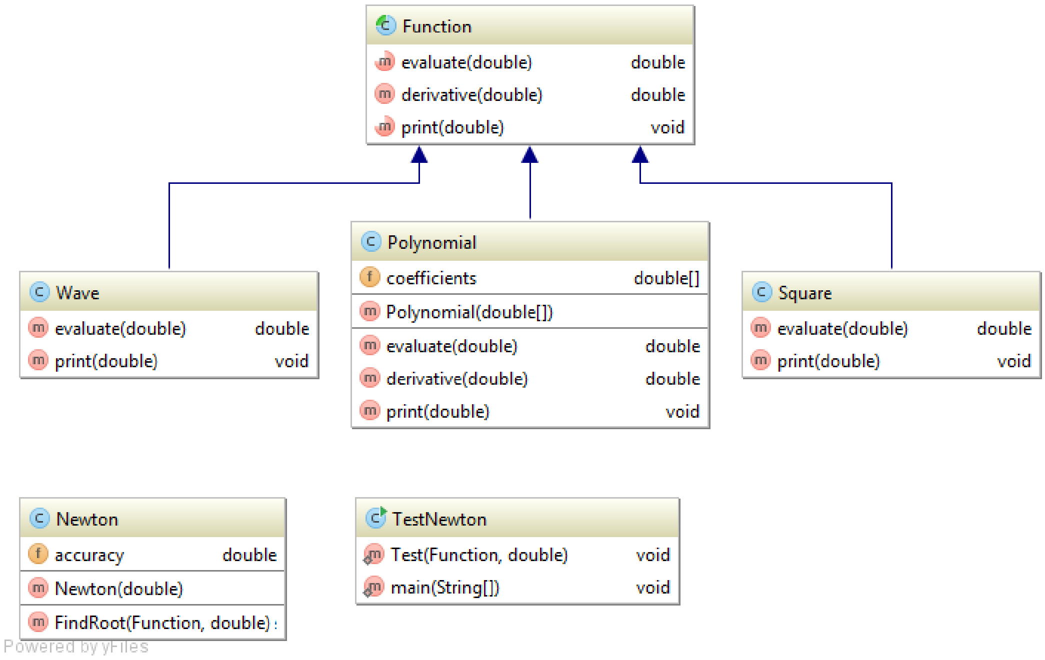

*Figure 1: Class Diagram of the final application you should implement*

## 1.2 class Function

The main focus of this exercise is to have you use and understand inheritance as you learned in class. The abstraction which is modeled through inheritance within this exercise is the fact, that we can cluster waves, polynoms and other mathematical functions. We do exactly so by sharing their common properties in an ***abstract*** parent class $Function$ which contains the following methods:

| name | parameters | return | remarks |
|---|---|---|---|
| evaluate | double | double | should be abstract since we do not know how to generically evaluate a function |
| derivative | double | double | calculates the derivative by using the passed input value and applying the following formula to it<br>$\frac{evaluate(input+0.00000001)-evaluate(input)}{0.00000001}$ |
| print | double | void | an abstract print method that will require every child to implement a print method |

Note that we are making the class and some of its member methods abstract. Making the class abstract allows us to ensure that only other classes that extend the $Function$ class can be passed, whenever a $Function$ instance is expected. (as can be seen in the given TestNewton code). It disallows the creation of a "pure" instance of $Function$. (e.g. "new Function" would give you a compile error)

## 1.3 class Square

This class should extend the abstract class $Function$. For this class we are happy with the default implementation of $derivative$ implemented in the parent class $Function$ and therefore do not need to reimplement it. (Codesharing through inheritance). We do however need to implement the two abstract methods of the parent class as follows:

| name | parameters | return | remarks |
|---|---|---|---|
| evaluate | double | double | return $input * input$ |
| print | double | void | `System.out.print("("+input+")^2");` |

## 1.4 class Wave

This class is implemented analogous to $Square$.

| name | parameters | return | remarks |
|---|---|---|---|
| evaluate | double | double | return $sin(input) + cos(input)$ |
| print | double | void | `System.out.print("sin(" +input+ ") + cos(" +input+ ")");` |

## 1.5 class Newton

The class $Newton$ implements the Newton method and consists only of the constructor, which sets a given accuracy (saved as a member variable accuracy) for the method (when to stop) and the Newton method itself which can be called with the $FindRoot$ function.

| name | parameters | return | remarks |
|---|---|---|---|
| Newton | double | - | the constructor of the class, simply sets the member variable $accuracy$ |
| FindRoot | Function, double | double | gets the desired function and a starting value passed. Then applies the Newton method till the value of the current location converges to a value that is smaller then $accuracy$. If this condition is met, it returns the current position (the root of the function found). |

## 1.6 class Polynomial

The class $Polynomial$ is a bit more involved then the other two classes that inherit from $Function$ but should still be easy enough to implement. It consists of the following parts

| name | parameters | return | remarks |
|---|---|---|---|
| Polynomial | double[] | - | the constructor of the class which takes an array of double numbers. Each number represents a Coefficient of a polynomial. (meaning it can have different degree depending on the length of the passed array). The constructor creates a local copy of the passed array in the member variable $coefficients$. The coefficents are passed in increasing order (e.g. the 0th element of the passed array is the 0th coefficent). |
| evaluate | double | double | calculates the resulting value of a polynomial of the form $coefficent_0+coefficent_1*input^1+coefficent_2*input^2+\ldots$ |
| derivative | double | double | calculates the derivative of a polynomial analytically and return it |
| print | double | void | custom print method that prints in a loop with<br>`print(coefficients[n] + " * (" + input + ")^" + n);` |