ETH zürich

# Assignment 9

Felix Friedrich, Lars Widmer
TA lecture, Informatics II D-BAUG

April 19, 2014

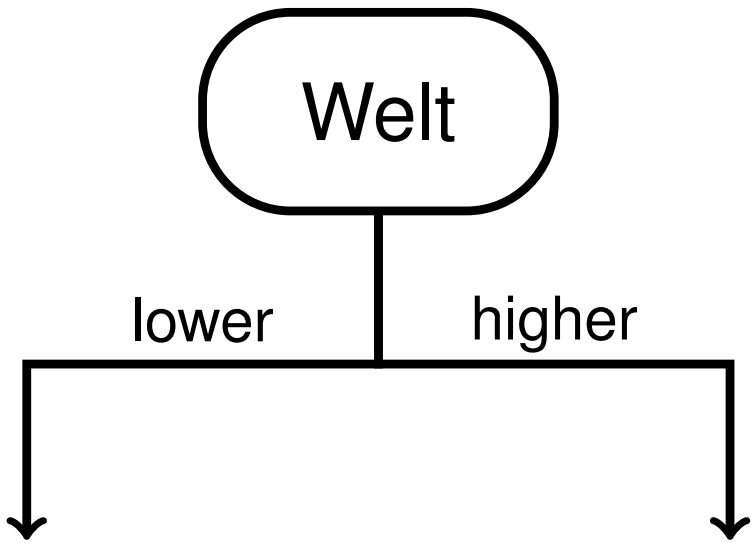
# “Präsenzstunden” Today

## In the **standard** room

- **HIL E15.2**
- **15:00 - 17:00** *(official: 16:00)*
- **Timon Gehr** *(arriving 15:45)*
- **Lei Zhong** *(arriving 15:00)*

# Outline

# Tree Element

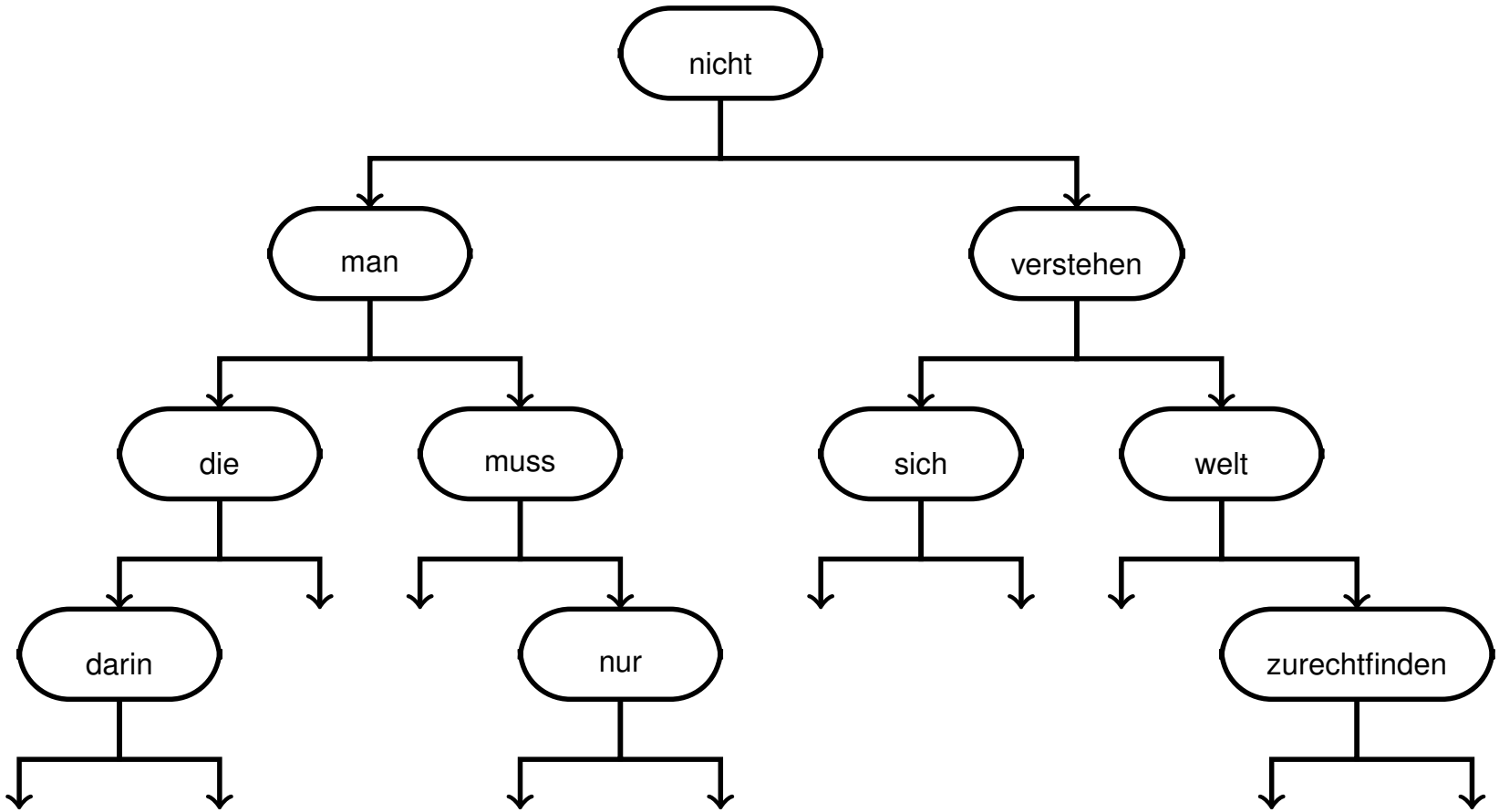



We build the **data structure** of a **tree** using **elements** as depicted on the lefthandside.

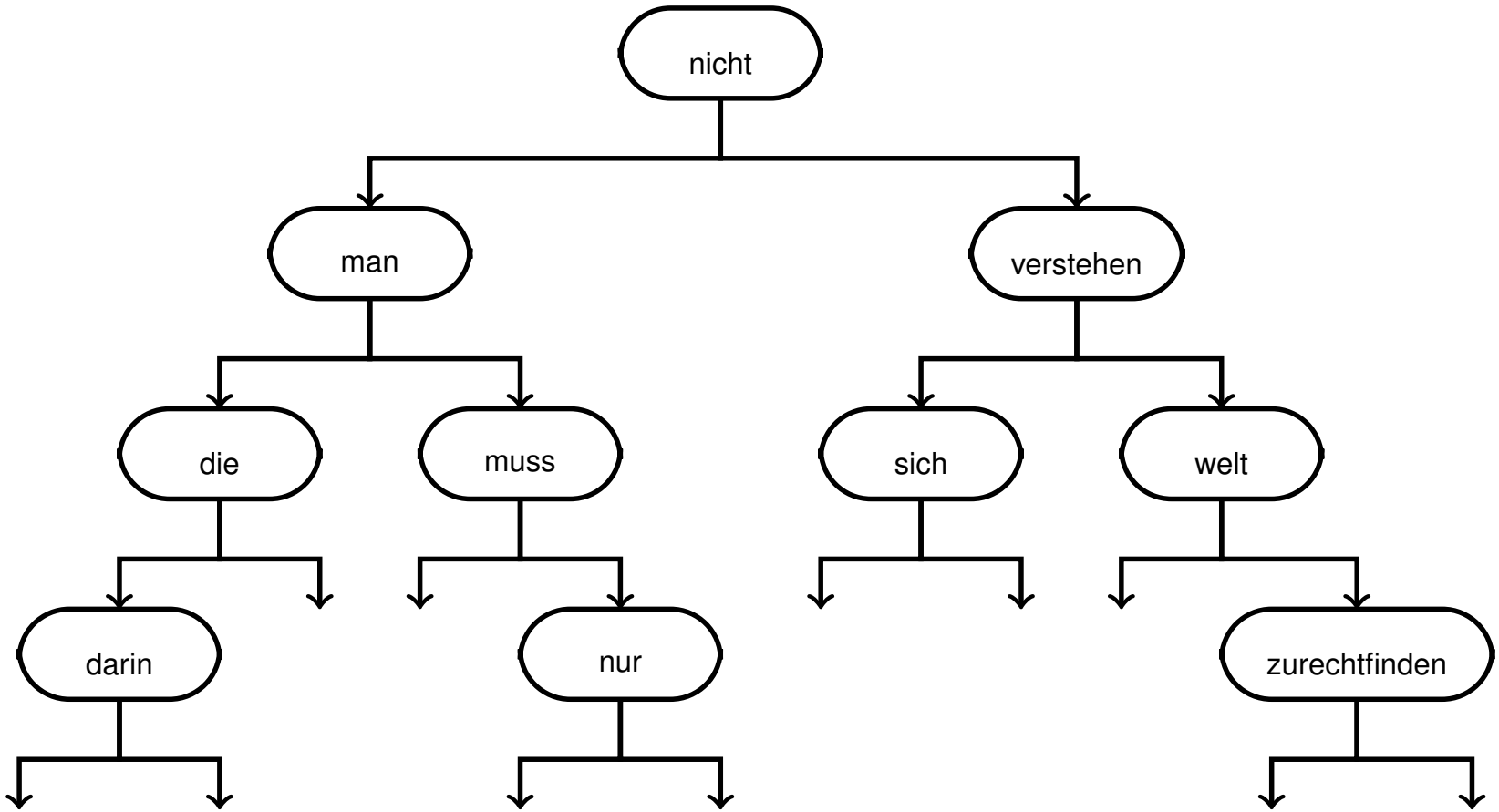
Same as in a linked list, **references** are used to **connect** the data structure.

In a **binary tree**, every element has **two** references.

# Binary Tree Example

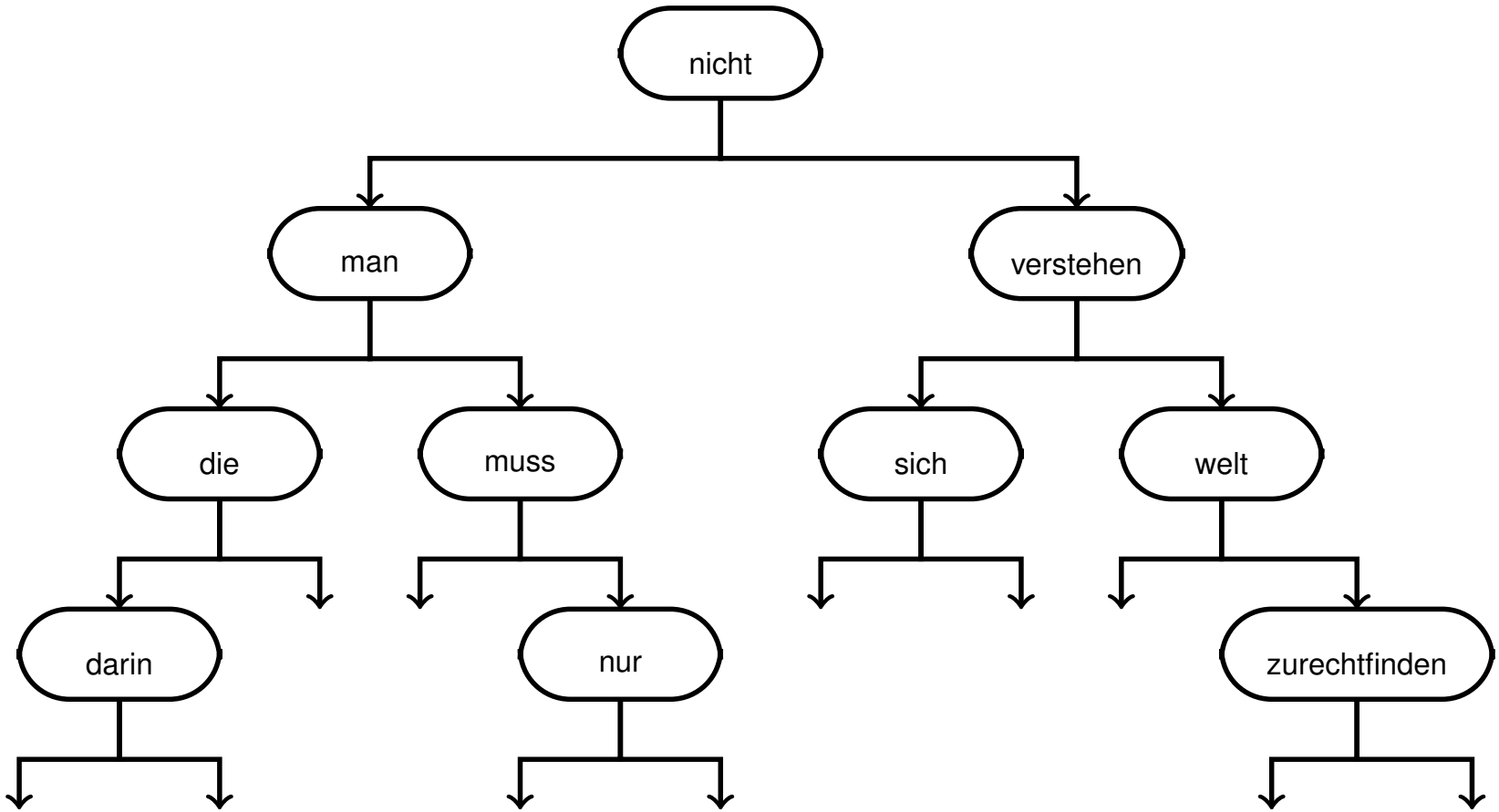


# Binary Tree Example



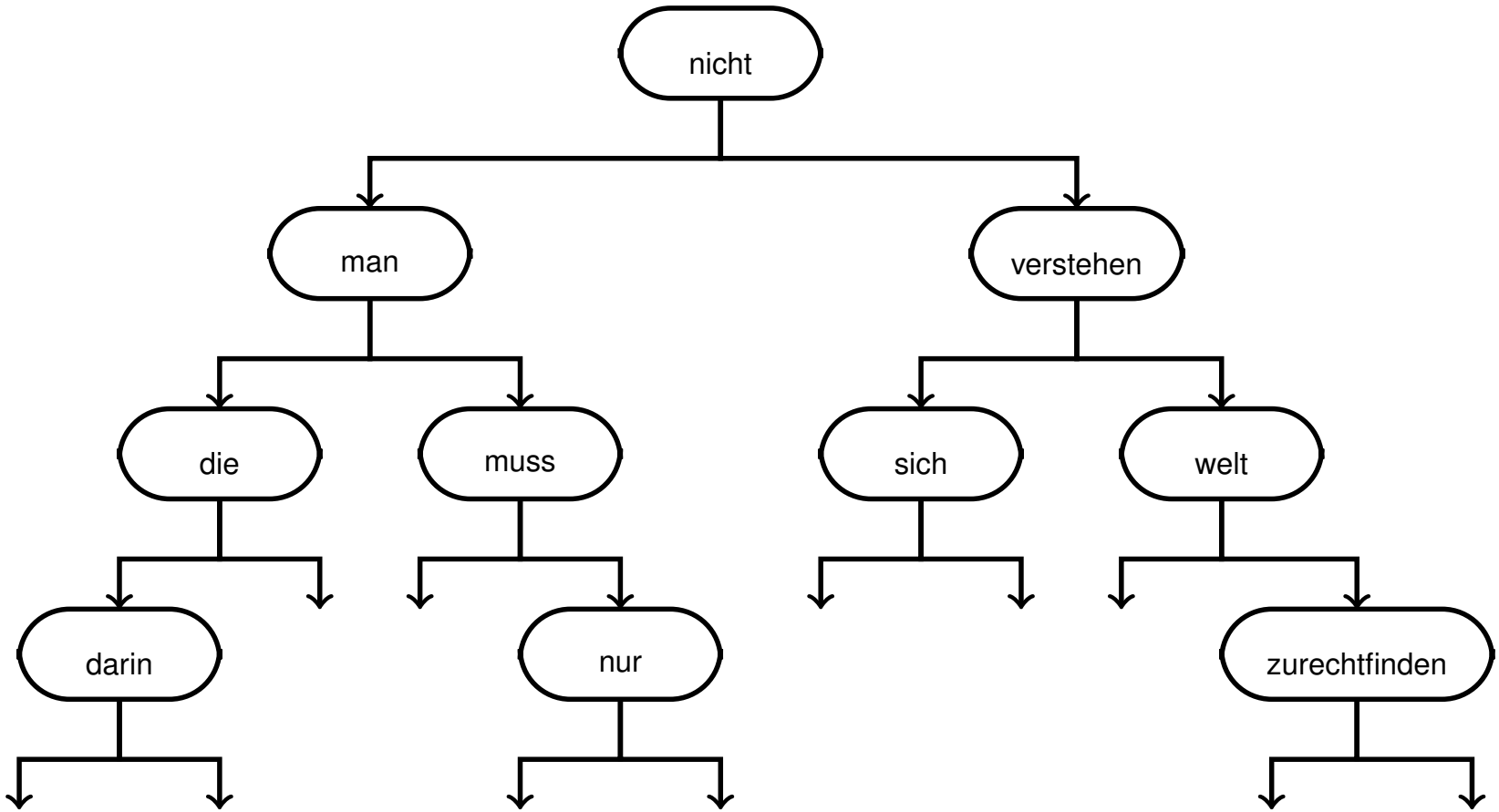

Question: **Which word was inserted first?**

# Binary Tree Example



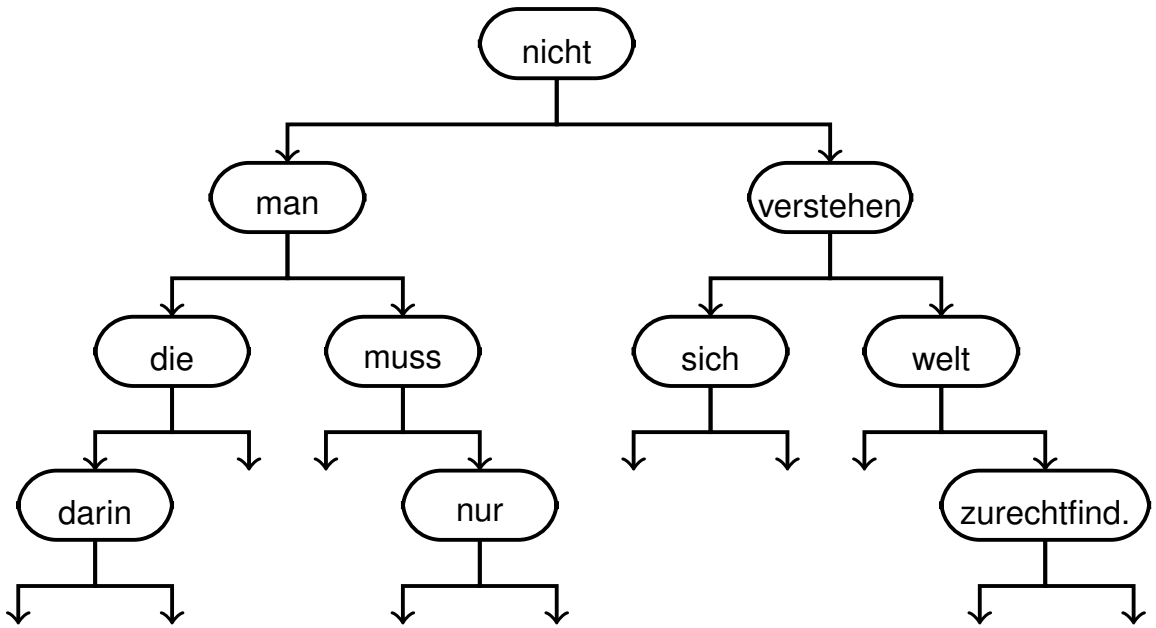

Question: **Which word was inserted first?** $\rightarrow$ *"nicht"*

# Binary Tree Example



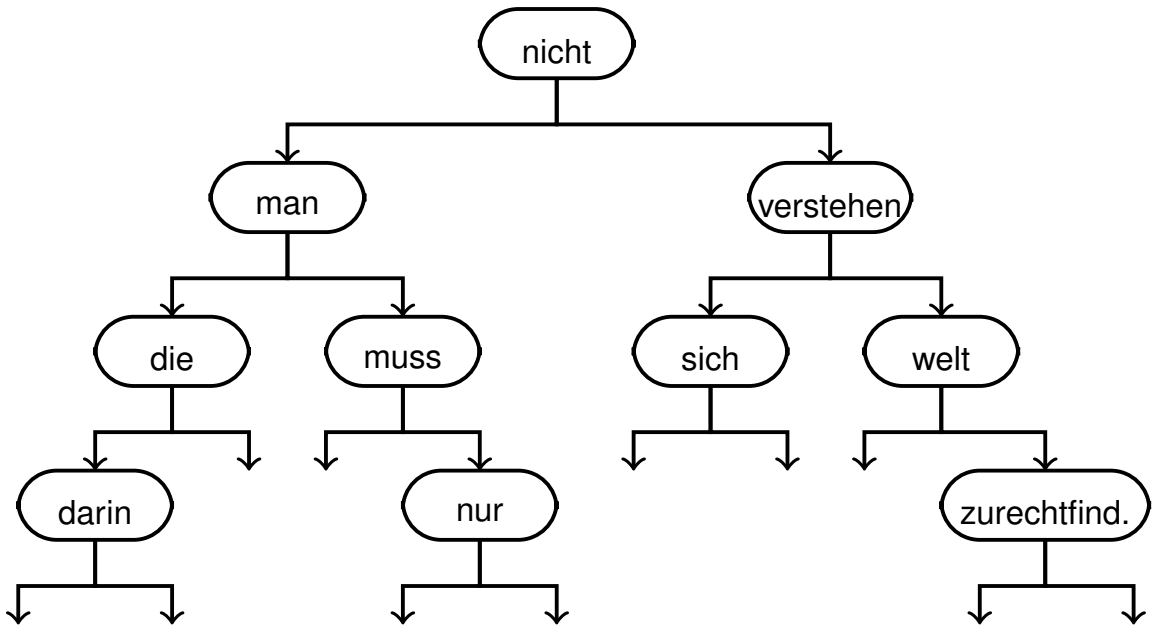

Question: **Which word was inserted first?** → *"nicht"* → **Why?**

# Binary Tree




We start with an empty tree. New elements are always connected where a `null`-reference was before. Thus *"nicht"* must have been the first word inserted.

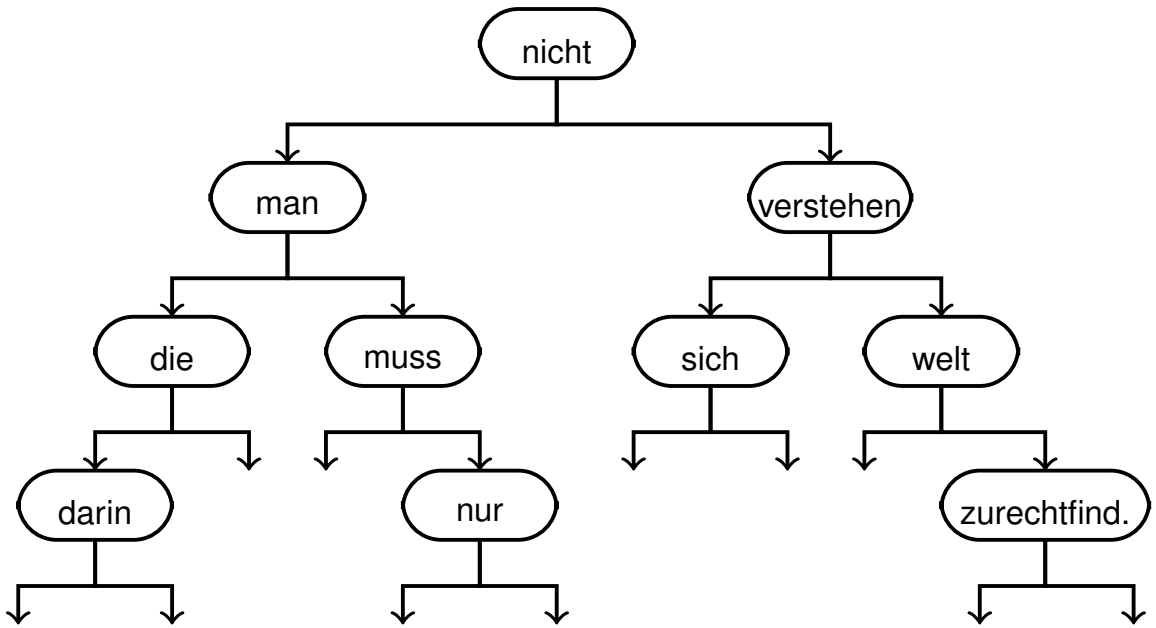
# Binary Tree




We start with an empty tree. New elements are always connected where a `null`-reference was before. Thus *"nicht"* must have been the first word inserted.

Question: **Where would the word *"einstein"* be put?**

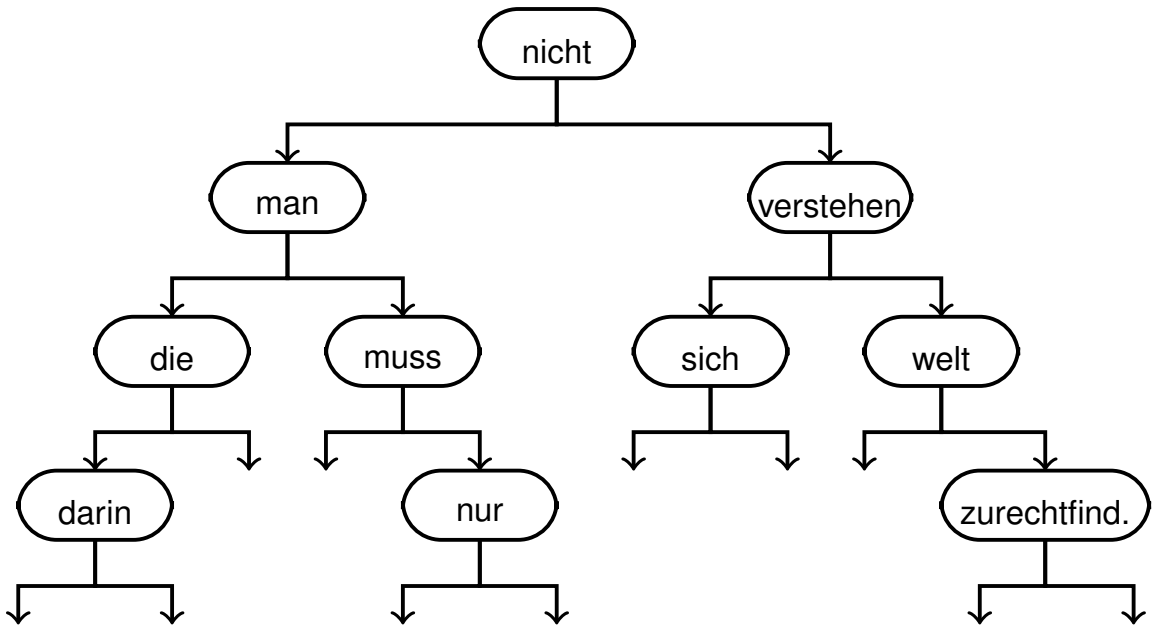
# Binary Tree




We start with an empty tree. New elements are always connected where a `null`-reference was before. Thus *"nicht"* must have been the first word inserted.

Question: **Where would the word *"einstein"* be put?**
$\rightarrow$ To the righthandside reference of *"die"*.

# Binary Tree




We start with an empty tree. New elements are always connected where a `null`-reference was before. Thus *"nicht"* must have been the first word inserted.

Question: **Where would the word *"einstein"* be put?**
$\rightarrow$ To the righthandside reference of *"die"*. $\rightarrow$ **Why?**

# Binary Tree

## Where to put “einstein”: **Traversing a Tree**

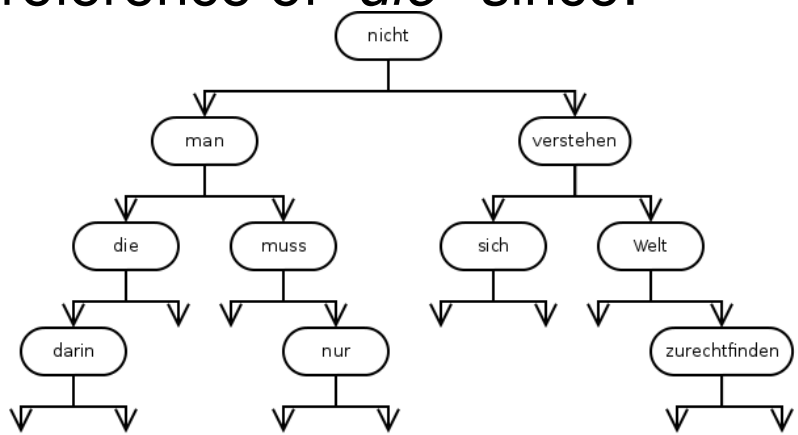
**Searching** and **inserting** in a binary tree are quite similar.
We always follow the references `lower` *(left)* or `higher` *(right)*, according to the comparison.

# Binary Tree

## Where to put “einstein”: **Traversing a Tree**

**Searching** and **inserting** in a binary tree are quite similar. We always follow the references `lower` *(left)* or `higher` *(right)*, according to the comparison.

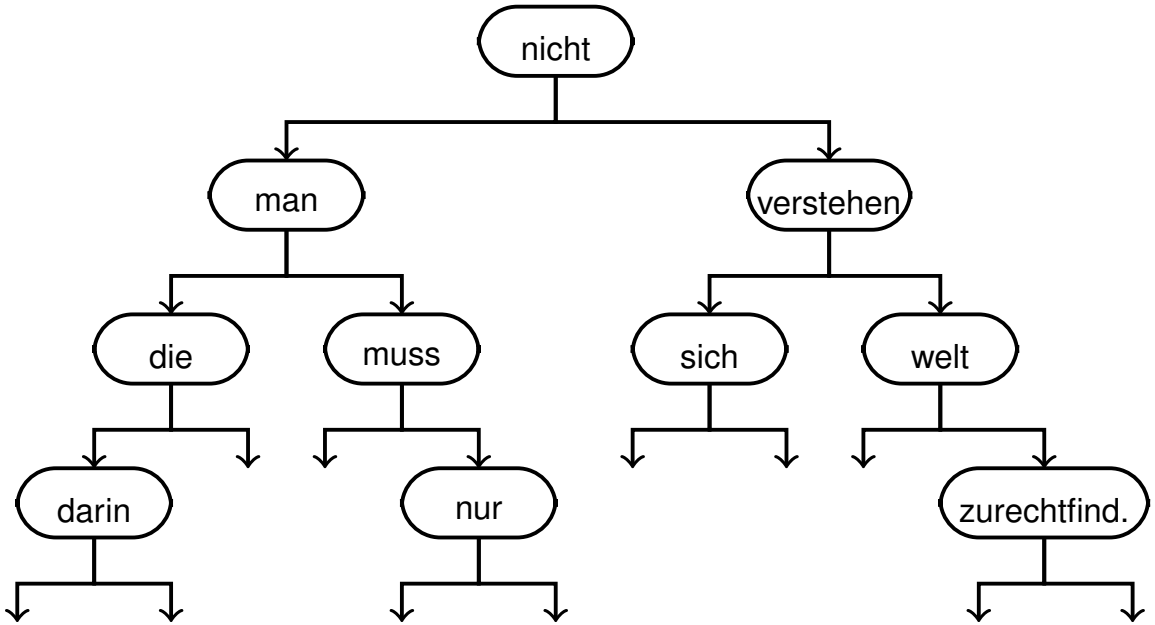
So, the new element *“einstein”* would go to the righthandside reference of *“die”* since:




- *“einstein”* $<$ *“nicht”*
- *“einstein”* $<$ *“man”*
- *“einstein”* $>$ *“die”*
- Reference `higher` of *“die”* is `null`

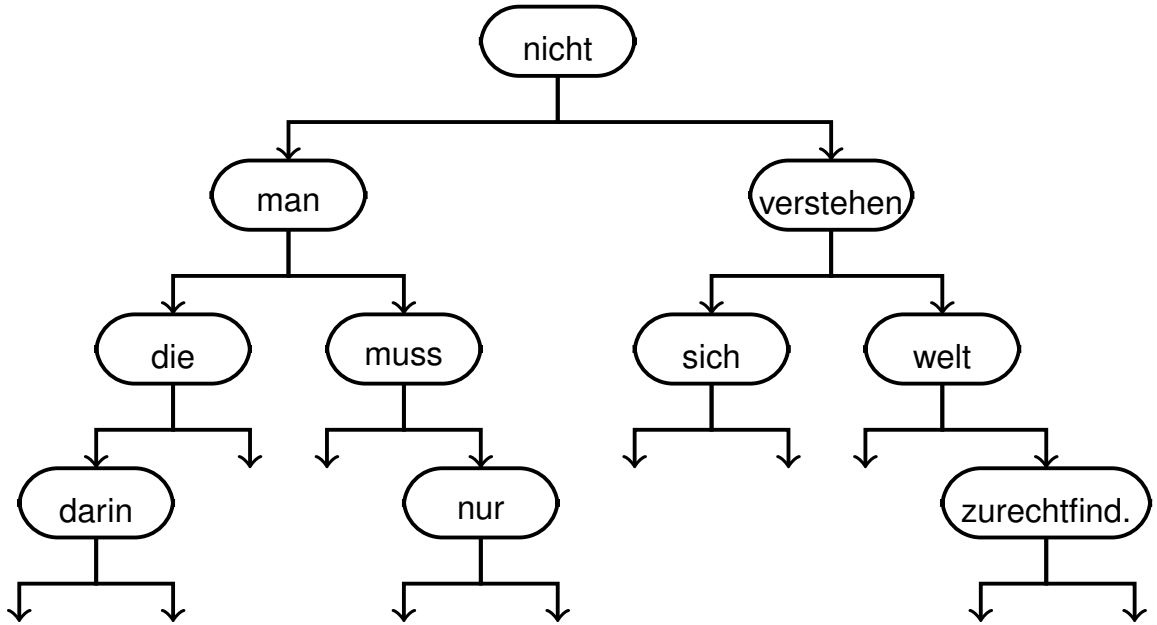
# Binary Tree

Where would the word *“Albert”* be put?



# Binary Tree

Where would the word *“Albert”* be put?




→ Exactly, to the bottem left `null`-reference, `lower` of *“darin”*.

# Outline

# Exception Handling

```
1 try {
2     // ... file access code
3 } catch (IOException e) {
4     e.printStackTrace();
5 }
```

**try-catch**

Many things can go wrong when working with files. E. g. the file can be write protected.

# Exception Handling

```
1 try {
2     // ... file access code
3 } catch (IOException e) {
4     e.printStackTrace();
5 }
```

**try-catch**

Many things can go wrong when working with files. E. g. the file can be write protected.
We therefore have to deal with eventual exception. We do this by surrounding the code for file handling with `try-catch`.

# Java `FileWriter`

```
1 BufferedWriter out =
2         new BufferedWriter(
3         new FileWriter("highscore.txt"));
```

## Writer

You can blindly reuse this given code.

# Java `FileWriter`

```
1 BufferedWriter out =
2         new BufferedWriter(
3         new FileWriter("highscore.txt"));
```

## Writer

You can blindly reuse this given code.
We use a predefined `FileWriter` which does all the low-level output for us. For performance reasons we wrap the `FileWriter` into a `BufferedWriter`.

# Java `FileWriter`

```
1 BufferedWriter out =
2         new BufferedWriter(
3         new FileWriter("highscore.txt"));
```

## Writer

The `BufferedWriter` does some sort of **caching** for us. This means, it doesn't immediatley write every single character to disk when we write to the `BufferedWriter` but rather **collects the content** and finally writes the file efficiantly in one go.

# Java `FileReader`

```
1 BufferedReader in =
2         new BufferedReader(
3         new FileReader("highscore.txt"));
```

## Reader

Same here: You can blindly reuse this given code.

# Java `FileReader`

```
1 BufferedReader in =
2         new BufferedReader(
3         new FileReader("highscore.txt"));
```

## Reader

Same here: You can blindly reuse this given code.
We use a predefined `FileReader` which does all the low-level input for us. For performance reasons we wrap the `FileReader` into a `BufferedReader`. The `BufferedReader` does some sort of **caching** for us. This means, it doesn't necessarly read from disk when we read from the `BufferedReader` but rather **pre-reads the file in advance**.

# Writing to File

```
1 out.write(int i);
2 out.write(String str);
3 out.write('\n'); // CR/LF character
```

## **Writing** to File

Writing to the file now is easy. The `write`-method is overloaded for many different parameter types.

# Writing to File

```
1 out.write(int i);
2 out.write(String str);
3 out.write('\n'); // CR/LF character
```

## **Writing** to File

Writing to the file now is easy. The `write`-method is overloaded for many different parameter types.
Internally the file has some sort of a pointer which advances with every `write`. So you don't overwrite preceding `write`s, but you add up to them. Simple: `write(‘‘H’’); write(‘‘o’’); write(‘‘i’’);` and `write(‘‘Hoi’’);` produce the **same** result!

# Reading from File

```
1 int i = in.read();
2 String str = in.readLine();
```

## **Reading** from File

The operation `read` reads a single character *(unicode)* from the file and interprets it as an `int`.

# Reading from File

```
1 int i = in.read();
2 String str = in.readLine();
```

## **Reading** from File

The operation `read` reads a single character *(unicode)* from the file and interprets it as an `int`.
While `readLine` reads **from the current position of the file pointer** until the end of the line (’\n’).

# Outline

# Implementing `BinaryTree`

**`BinaryTree`**

We didn't give you any details on how to implement the required functions. We know that you're smart and we believe that your experience is sufficient to solve the assignment from scratch.

# Implementing `BinaryTree`

## `BinaryTree`

We didn't give you any details on how to implement the required functions. We know that you're smart and we believe that your experience is sufficient to solve the assignment from scratch.

## Important Information

*There's (at least) one suboptimal point in the assignment . . .* In exercise 9.1 i) only **insert** the **numbers** 100 **to** 999 into the tree. Don't start at 1! Otherwise the **result** becomes a bit **unclear**.

# Outline

# Extending Highscore

```
1 protected LinkedList<Player> highscore;
```

## Creating a **Subclass**

With “extending” we refer to **inheritance** in the sense of **OOP**. Therefore the methods `load` and `store` go into a subclass of `Highscore`.

# Extending Highscore

```
1 protected LinkedList<Player> highscore;
```

## Creating a **Subclass**

With “extending” we refer to **inheritance** in the sense of **OOP**. Therefore the methods `load` and `store` go into a subclass of `Highscore`. The only thing you have to change in your `class Highscore` is to make the internal **field** *(storage)* `highscore` **accessible for subclasses**.

# Extending Highscore

```
1 protected LinkedList<Player> highscore;
```

## Creating a **Subclass**

With “extending” we refer to **inheritance** in the sense of **OOP**. Therefore the methods `load` and `store` go into a subclass of `Highscore`. The only thing you have to change in your `class Highscore` is to make the internal **field** *(storage)* `highscore` **accessible for subclasses**. Fields which are `private` are inaccessible in subclasses while `protected` ones can be used as if they were defined in the same `class`.

# Closing Files finally

```
1 try {
2     BufferedWriter out = new BufferedWriter(
3             new FileWriter("highscore.txt"));
4     // ...
5     out.close();
6 } catch (IOException e) {
7     e.printStackTrace();
8 }
```

## Closing the File

As you can see on line 6, you have to **close the file**, when you're done with writing. Otherwise Eclipse will raise an error/warning.

# Closing Files finally

```
1 BufferedReader in = new BufferedReader(
2         new FileReader("highscore.txt"));
3 // ...
4 in.close();
```

## Closing the File

The same applies when reading the file. Always close the file, when you're done.

# Closing Files finally

```
1 BufferedReader in = new BufferedReader(
2         new FileReader("highscore.txt"));
3 //  ...
4 in.close();
```

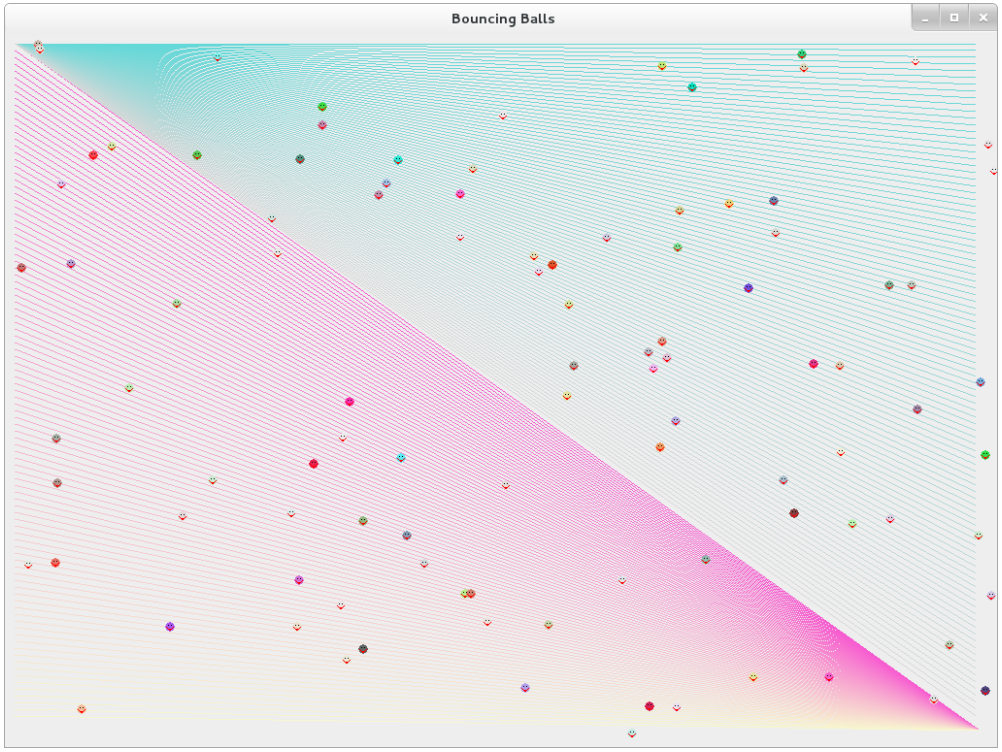
## Closing the File

The same applies when reading the file. Always close the file, when you're done.

... It's never wrong to **avoid** keeping files open for the full program execution. Better prepare everything, open the file, do the file I/O and close it again.

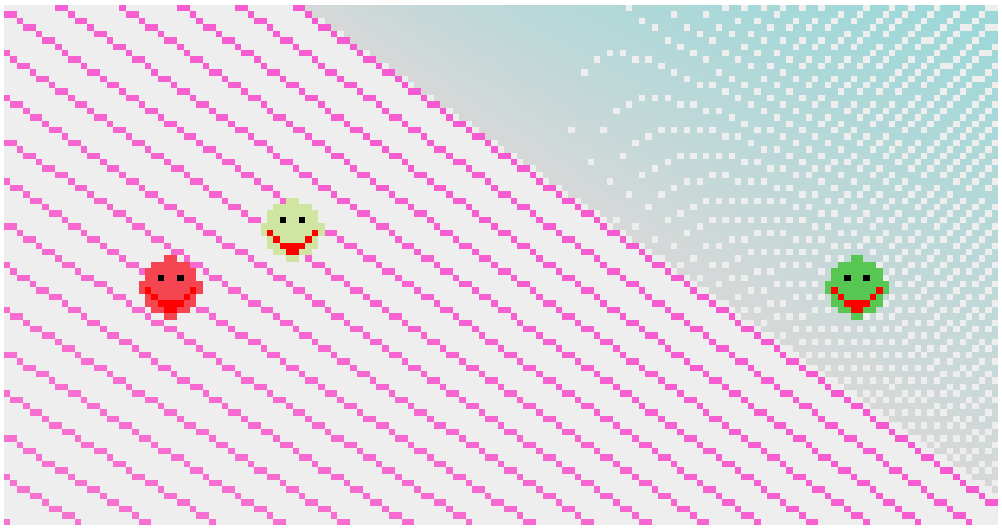
# Outline

# Bouncing Smileys!



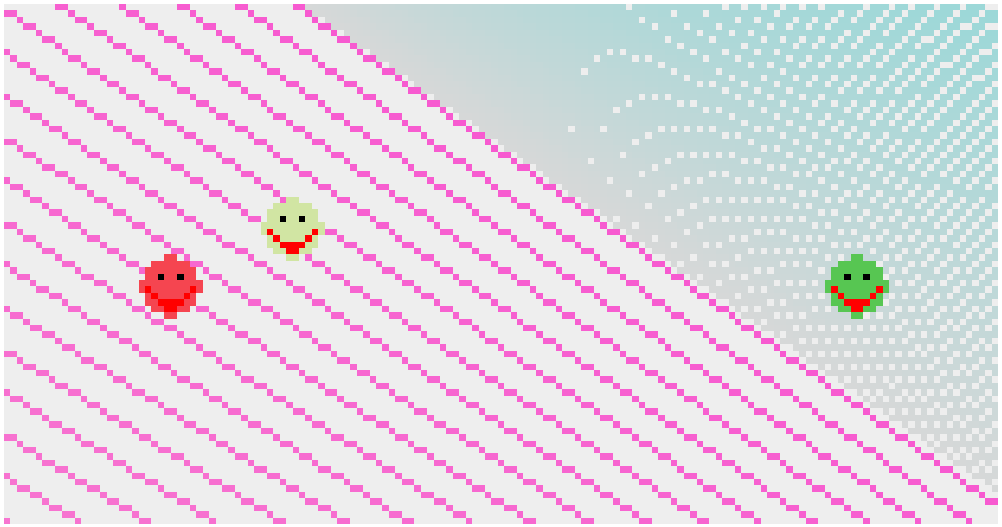

For the third exercise of the assignment, you may implement whatever you wish. Especially you may have some fun with Graphics. Like the example in the image.

# Bouncing Smileys!



They aren't just circles but actual Smileys, driving through the image diagonally and bouncing off the borders.

# Bouncing Smileys!



## Good Luck!

. . . But we're sure, you'll have better ideas ;-).

*Be creative & enjoy*

# Outline

# Matlab Function

```
1 function res = f(x)
2     res = sqrt(x^6-x^3+12) /
3           log((x^2+4)*(x^2-2)+10);
4 end
```

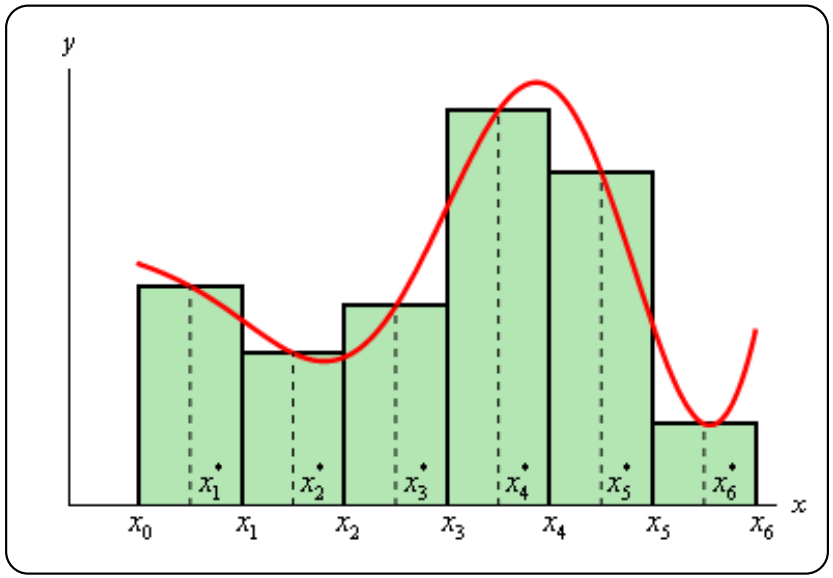
**That's it**

Simple, isn't it?

# Outline

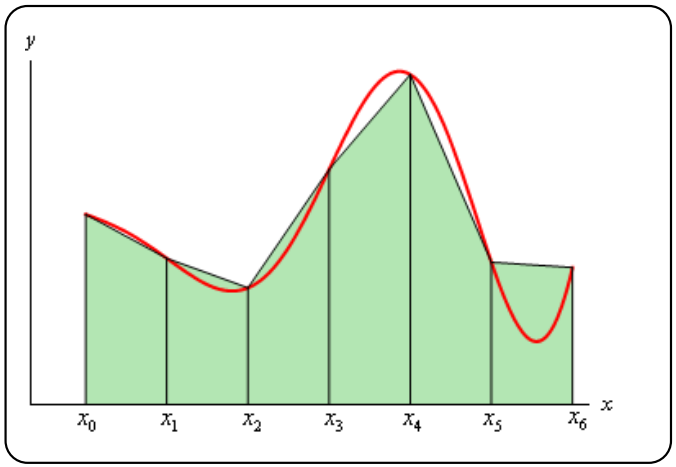
# Rectangle Rule in Matlab

```
1  function res = int_rechteck(f, a, b, steps)
2      step = (b-a)/steps;
3      res = 0;
4      pos = a+step/2;
5      for i = 1:steps
6          res = res + f(pos);
7          pos = pos + step;
8      end
9      res = res * step;
10 end
```



# Trapezoidal Rule in Matlab

```
1  function [res] = int_trapez(f, a, b, steps)
2      step = (b-a)/steps;
3      res = 0;
4      pos = a;
5      fn = f(pos);
6      for i = 1:steps
7          pos = pos + step;
8          fm = f(pos);
9          res = res + (fn + fm) / 2;
10         fn = fm;
11     end
12     res = res * step;
13 end
```



# Outline

# Measuring Time in Matlab

## Measuring Time

In Matlab we use the following code to measure the execution time of our functions:

```
1 tic
2 rechteck = int_rechteck(@f, -0.4, 0.4, 100000)
3 toc
4
5 tic
6 trapez = int_trapez(@f, -0.4, 0.4, 100000)
7 toc
```

# Duration Matlab Calculation

## Results

The results show that the rectangle rule is a bit faster, most probably because it's computationally a little bit cheaper.

```
1 rechteck =
2     -0.0170
3 Elapsed time is 0.059683 seconds.
4
5 trapez =
6     -0.0170
7 Elapsed time is 0.062395 seconds.
```

# The same Function in Java

## Java Function

Now we need the same function in Java.

```
1 public double f(double x) {
2     if (x==0) {
3         return 0;
4     } else {
5         return Math.sin(1/x)*x;
6     }
7 }
```

# Rectangle Rule in Java

The rectangle rule can be implemented e. g. like this:

```
1  public double int_rechteck(double a,
2                             double b, int steps) {
3      double step = (b-a)/steps;
4      double res = 0;
5      double pos = a+step/2;
6      for (int i=0; i<steps; ++i) {
7          res = res + f(pos);
8          pos = pos + step;
9      }
10     return res * step;
11 }
```

# Trapezoidal Rule in Java

```
1  public double int_trapez(double a,
2                           double b, int steps) {
3      double step = (b-a)/steps;
4      double res = 0;
5      double pos = a;
6      double fn = f(pos);
7      for (int i=0; i<steps; ++i) {
8          pos = pos + step;
9          double fm = f(pos);
10         res = res + (fn + fm) / 2;
11         fn = fm;
12     }
13     return res * step;
14 }
```

# Java Test Code

To determine the duration of the calculation in Java we use the following code:

```
1  public static void main(String[] args) {
2      TestFunction tf = new TestFunction();
3      long time = System.currentTimeMillis();
4      double r = tf.int_rechteck(-0.4, 0.4, 100000);
5      long dr = System.currentTimeMillis() - time;
6      time = System.currentTimeMillis();
7      double t = tf.int_trapez(-0.4, 0.4, 100000);
8      long dt = System.currentTimeMillis() - time;
9      System.out.println(r+": "+dr+"ms");
10     System.out.println(t+": "+dt+"ms");
11 }
```

# Matlab vs. Java

## Commandline Output

```
1 -0.016978080396639117: 9ms
2 -0.016977882426847 82: 8ms
```

What do you think? ...

# Matlab vs. Java

## Commandline Output

```
1 -0.016978080396639117: 9ms
2 -0.016977882426847 82: 8ms
```

What do you think? ...

## Results

The calculations come up with the same result.
**But Java is around** $7\times$ **faster!??** *... so far*

# Vectorization: Optimizing Matlab Solution

```
1  function res = int_rechteck_v(f, a, b, N)
2      x = linspace(a,b,N+1);
3      res = (b-a)/N * sum(f((x(1:end-1)
4                             + x(2:end))/2));
5  end
6
7  function res = int_trapez_v(f, a, b, N)
8      x = linspace(a,b,N+1);
9      res = (b-a)/(2*N) * sum(f(x(1:end-1))
10                             + f(x(2:end)));
11 end
```

# Vectorization: Optimizing Matlab Solution

```
1  function res = int_rechteck_v(f, a, b, N)
2      x = linspace(a,b,N+1);
3      res = (b-a)/N * sum(f((x(1:end-1)
4                             + x(2:end))/2));
5  end
6
7  function res = int_trapez_v(f, a, b, N)
8      x = linspace(a,b,N+1);
9      res = (b-a)/(2*N) * sum(f(x(1:end-1))
10                             + f(x(2:end)));
11 end
```

Using this code, Java and Matlab are quite similar in speed. Matlab is even a little bit faster ($4ms$, $6ms$).

# Outline

# Bubblesort in Matlab

```
1  function res = bubbleSort(vec)
2      done = 0;
3      while done == 0
4          done = 1;
5          for n = 1:length(vec)-1
6              if vec(n) < vec(n+1)
7                  t = vec(n);
8                  vec(n) = vec(n+1);
9                  vec(n+1) = t;
10                 done = 0;
11             end
12         end
13     end
14     res = vec;
15 end
```

# Calling `bubbleSort`

## Two Ways:

What's the **difference**?

- Call as a Procedure

```
1 bubbleSort(v);
```

- Call as a Function

```
1 v = bubbleSort(v);
```

# Calling `bubbleSort`

## Two Ways:

What's the **difference**?

- Call as a Procedure

```
1 bubbleSort(v);
```

- Call as a Function

```
1 v = bubbleSort(v);
```

Matlab always does **call by value**. This means the parameter values are basically copied for the procedure. Therefore all the changes within a procedure don't affect the original data.
But having `bubbleSort` as a function allows to read back the result of the sorting.

# Feel free ...

**Please**

- Questions?
- Feedback?
- Additions?
- Remarks?
- Wishes?
- ...

# All the Best!

***We wish you happy easter and nice holidays!***


*Chahat, Fabian, Fabian, Lei, Nico, Oskar, Raffaele, Renzo, Robin, Sandro, Severin, Simon, Temmy, Timon, Urs, Felix Friedrich and Lars*