

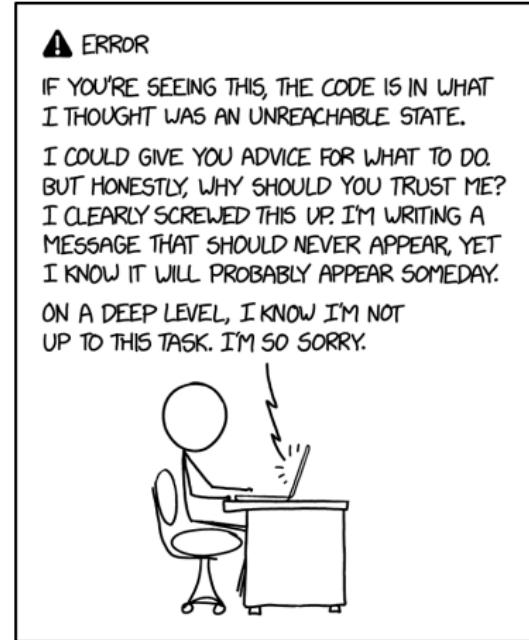


Exceptions, Rekursion

Vorkurs Informatik



Getting started ...



NEVER WRITE ERROR MESSAGES TIRED.

©CC-BY-NC Randall Munroe, <https://xkcd.com/2200/>

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

(nach Folien von L. Vettin, W. Kessler)



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Licence

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

Motivation

- Wenn wir bisher von “Fehlern” gesprochen haben, waren es meist syntaktische Fehler (“Schreibfehler”):
`iff (1 < 2) tehn system.out.println("Hello")`
- Es gibt aber auch Fehler, die erst auftreten, wenn das Programm bereits läuft:
 - Versuch eine Datei zu öffnen, die nicht existiert.
 - Division durch 0.
 - Eingabe eines Textes durch einen Nutzer, obwohl eine Zahl gefragt ist.
 - ...
- Diese Fehler können nicht vom Compiler gefunden werden.
- Tritt ein Fehler auf, bricht die Ausführung des Programms ab.

Motivation

```
public class Test {  
  
    public static void main(String[] args) {  
  
        double c = 5/0;  
        System.out.println(c);  
    }  
  
}
```

Abbruch mit Fehlermeldung:

```
Exception in thread "main"  
java.lang.ArithmetricException: / by zero  
at Test.main(Test.java:5)
```

Motivation

- Wenn wir eigene Methoden schreiben, möchten wir evtl. verhindern, dass diese mit bestimmte Eingaben aufgerufen werden, da es sonst zu Fehlern kommt.
- Beispiel:

```
public int factorial(int n) {  
    int result=1;  
    for( int i=1; i<=n; i++){ result = result*i; }  
    return result;  
}
```

- `factorial(-1);` ist nicht definiert
- `factorial(200);` verursacht einen Integer-Overflow.

Motivation

- Erste Idee: Wir führen explizite Fehlerwerte ein.

```
public int factorial(int n) {  
  
    int result=1;  
    if(n < 0 || n > 20){  
        result = -42;  
    } else {  
        for( int i=1; i<=n; i++){ result = result*i; }  
    }  
    return result;  
}
```

Motivation

- Erste Idee: Wir führen explizite Fehlerwerte ein.

```
public int factorial(int n) {  
  
    int result=1;  
    if(n < 0 || n > 20){  
        result = -42;  
    } else {  
        for( int i=1; i<=n; i++){ result = result*i; }  
    }  
    return result;  
}
```

- ⇒ Problem: Der Fehler wird dem Nutzer evtl. nicht bewusst (es könnte ja mit dem Ergebnis -42 weitergerechnet werden).
- ⇒ Problem: Verschlechterung der Programmstruktur durch Vermischung von Programmlogik und Fehlerkorrektur.

Motivation

- Erste Idee: Wir führen explizite Fehlerwerte ein.

```
public int factorial(int n) {  
  
    int result=1;  
    if(n < 0 || n > 20){  
        result = -42;  
    } else {  
        for( int i=1; i<=n; i++){ result = result*i; }  
    }  
    return result;  
}
```

- ⇒ Problem: Der Fehler wird dem Nutzer evtl. nicht bewusst (es könnte ja mit dem Ergebnis -42 weitergerechnet werden).
- ⇒ Problem: Verschlechterung der Programmstruktur durch Vermischung von Programmlogik und Fehlerkorrektur.
- Lösung: Wir machen dasselbe, was Java auch macht, z.B. bei Division durch 0 ...

Exceptions

Lösung:

- Exceptions / Ausnahmen
- Sie sind eine Möglichkeit zur Fehlererkennung und Fehlerbehandlung.
- Sie machen auftretende Abstürze im Programmverlauf sicher behandelbar.

Beispiel

- Wir werfen eine Exception, die beschreibt, um welchen Fehler es sich handelt:

```
public int factorial(int n) {  
  
    int result=1;  
    if(n < 0 || n > 20){  
        throw new IllegalArgumentException("n must be in range 0-20");  
    }  
    for( int i=1; i<=n; i++){ result = result*i; }  
    return result;  
}
```

Beispiel

- Wir werfen eine Exception, die beschreibt, um welchen Fehler es sich handelt:

```
public int factorial(int n) {  
  
    int result=1;  
    if(n < 0 || n > 20){  
        throw new IllegalArgumentException("n must be in range 0-20");  
    }  
    for( int i=1; i<=n; i++){ result = result*i; }  
    return result;  
}
```

- Zum Werfen einer Exception wird der Befehl `throw` verwendet.
- Man verwendet den Exception-Typ, der am besten zum Problem passt.
- Außerdem kann eine Fehlermeldung als String angegeben werden.

Exceptions

- Werden Exceptions nicht aufgefangen, bricht die Java Virtual Machine die Ausführung des Programms ab.

```
public class Test {  
  
    public static void main(String[] args) {  
  
        Calculator calc = new Calculator();  
        double result = calc.factorial(-1);  
    }  
}
```

Abbruch mit Fehlermeldung:

```
Exception in thread "main"  
java.lang.IllegalArgumentException: n must be in range 0-20  
at Calculator.factorial(Calculator.java:9)  
at Test.main(Test.java:7)
```

Exceptions

- Werden Exceptions nicht aufgefangen, bricht die Java Virtual Machine die Ausführung des Programms ab.
- Ein Java-Programm kann jedoch Exceptions “auffangen” und Maßnahmen einleiten, um mit dem Fehler umzugehen.
- Dazu schreibt man den Code, der die Exception werfen könnte, in einen `try`-Block.
- Nach dem `try`-Block kommt ein `catch`-Block, der Code für die Behandlung des Fehlers enthält.

Exceptions auffangen: try und catch

```
public class Test {  
  
    public static void main(String[] args) {  
  
        Calculator calc = new Calculator();  
  
        //Ask user for an int number  
        int a = ...;  
  
        try {  
            double result = calc.factorial(a);  
        } catch (IllegalArgumentException e) {  
  
            // Ask user for other number ...  
        }  
    }  
}
```

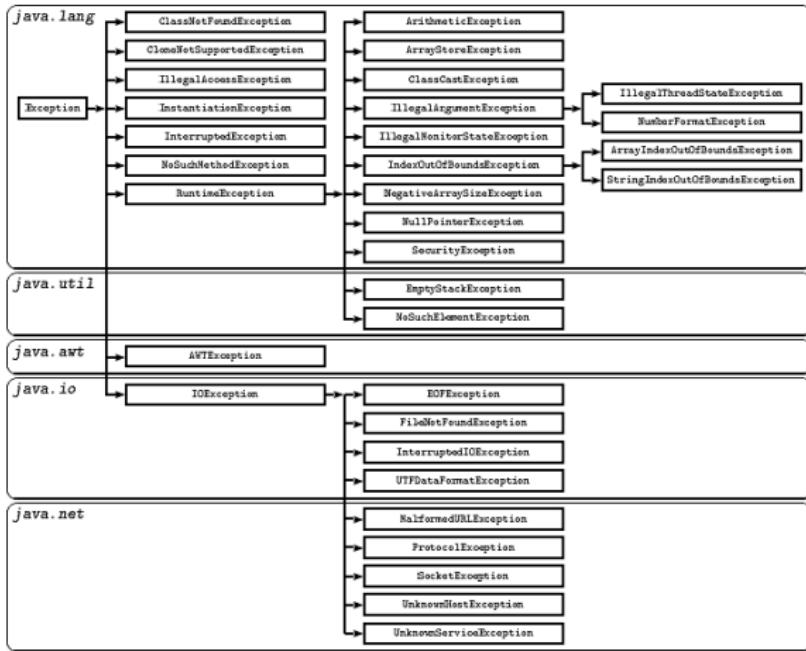
Syntax von try und catch

```
try {
    // Code, in dem potentiell Exceptions geworfen werden
    <Statement>;
    ...
} catch (ExceptionClass1 e) {
    // Abarbeitung von Exceptions des Typs ExceptionClass1
    <Statement>;
    ...
} catch (ExceptionClass2 e) {
    // Abarbeitung von Exceptions des Typs ExceptionClass2
    <Statement>;
    ...
}
```

- Es kann mehrere `catch`-Blöcke für verschiedene Typen von Exceptions geben.
- Ausgeführt wird der *erste* passende Block.

Exception-Hierarchie

Es gibt verschiedene Arten von Exceptions:



Quelle: http://www2.math.uni-wuppertal.de/~axel/skripte/oop/oop24_1.html

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

Rekursion

- Bei der Rekursion werden Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt.

Vorgehensweise

Eine rekursivee Lösung besteht aus zwei Teilen:

Rekursion

- Bei der Rekursion werden Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt.

Vorgehensweise

Eine rekursivee Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, lös es direkt (Basisfall).

Rekursion

- Bei der Rekursion werden Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt.

Vorgehensweise

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt (Basisfall).
- Andernfalls, teile es in einfachere Probleme und dann:
Löse die einfacheren Probleme auf dieselbe Art.

Beispiel: Teile eine Linie in 16 Teile

Beispiel: Teile eine Linie in 16 Teile

- Teile die Linie in 2 Teile.
- Das Problem wurde nun in zwei kleinere Teilprobleme zerlegt:
Teile jede Linie in 8 Teile.
- Wie löse ich die Teilprobleme?

Beispiel: Teile eine Linie in 16 Teile

- Teile die Linie in 2 Teile.
- Das Problem wurde nun in zwei kleinere Teilprobleme zerlegt:
Teile jede Linie in 8 Teile.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun haben wir 4 einfachere Teilprobleme.
- Wie löse ich die Teilprobleme?

Beispiel: Teile eine Linie in 16 Teile

- Teile die Linie in 2 Teile.
- Das Problem wurde nun in zwei kleinere Teilprobleme zerlegt:
Teile jede Linie in 8 Teile.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun haben wir 4 einfachere Teilprobleme.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun haben wir 8 einfachere Teilprobleme.
- Wie löse ich die Teilprobleme?

Beispiel: Teile eine Linie in 16 Teile

- Teile die Linie in 2 Teile.
- Das Problem wurde nun in zwei kleinere Teilprobleme zerlegt:
Teile jede Linie in 8 Teile.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun haben wir 4 einfachere Teilprobleme.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun haben wir 8 einfachere Teilprobleme.
- Wie löse ich die Teilprobleme? Mit derselben Methode:
Teile jede Linie in 2 Teile.
- Nun habe ich 16 Teile und bin fertig!

Beispiel: Generelles Vorgehen

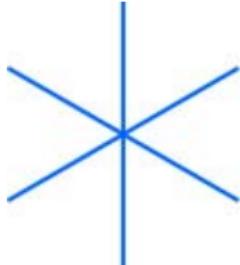
Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt.
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Linienteilung:

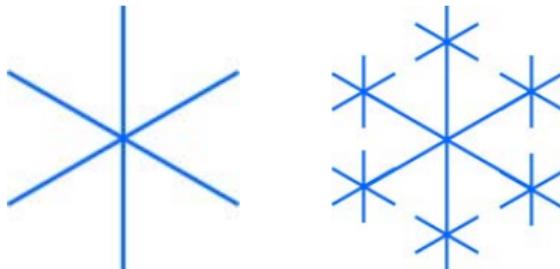
- Wenn ich so viele Teile habe wie ich will, stoppe.
- Andernfalls, teile die Linie in 2 Teile, dann:
 - Wende denselben Prozess auf jeden Teil an.

Beispiel: Schneeflocken zeichnen



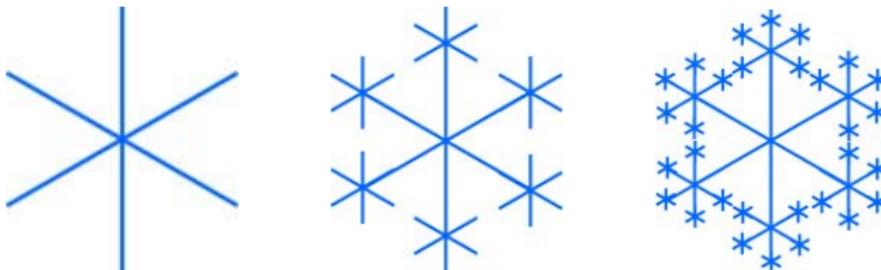
- Zeichne einen Stern mit 6 Linien.

Beispiel: Schneeflocken zeichnen



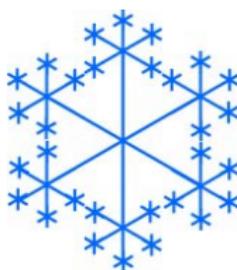
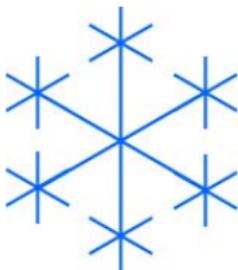
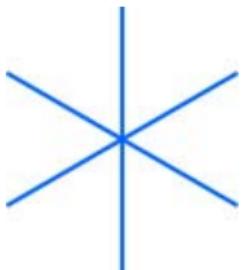
- Zeichne einen Stern mit 6 Linien.
- An das Ende jeder Linie, zeichne einen kleineren Stern.

Beispiel: Schneeflocken zeichnen



- Zeichne einen Stern mit 6 Linien.
- An das Ende jeder Linie, zeichne einen kleineren Stern.
- An das Ende jeder Linie, zeichne einen kleineren Stern.

Beispiel: Schneeflocken zeichnen



...

- Zeichne einen Stern mit 6 Linien.
- An das Ende jeder Linie, zeichne einen kleineren Stern.
- An das Ende jeder Linie, zeichne einen kleineren Stern.
- ...
- So lange bis die Sterne nicht mehr sichtbar sind.

Aufgabe: Rekursion

Überlegen Sie sich eine rekursive Lösung, um ...

- ein squareisches Blatt Papier in 64 kleine squaree zu falten.
- ein 1000-seitiges Buch zu lesen.
- eine 90-minütige langweilige Vorlesung durchzustehen.
- einen Liter Saft zu trinken.
- ein komplettes Aufgabenblatt zu bearbeiten.
- ...

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

Rekursion (allgemein)

Als Rekursion bezeichnet man den abstrakten Vorgang, dass Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt werden.

Rekursion

Rekursion (allgemein)

Als Rekursion bezeichnet man den abstrakten Vorgang, dass Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt werden.

Rekursion (Informatik)

In der Informatik (und Mathematik) bedeutet Rekursion, dass eine Funktion in ihrer Definition selbst nochmals aufgerufen wird.

Rekursion

Rekursion (allgemein)

Als Rekursion bezeichnet man den abstrakten Vorgang, dass Regeln auf ein Produkt, das sie hervorgebracht haben, von neuem angewandt werden.

Rekursion (Informatik)

In der Informatik (und Mathematik) bedeutet Rekursion, dass eine Funktion in ihrer Definition selbst nochmals aufgerufen wird.

Jedes iterativ lösbar Problem lässt sich auch rekursiv lösen!

Summe der natürlichen Zahlen bis n

$$\sum_{i=1}^n i$$

Summe der natürlichen Zahlen bis n

$$\sum_{i=1}^n i$$

Summe von 1 bis n : $n + \sum_{i=1}^{n-1} i$

Summe von 1 bis $n - 1$: $(n - 1) + \sum_{i=1}^{n-2} i$

Summe von 1 bis $n - 2$: $(n - 2) + \sum_{i=1}^{n-3} i$

...

Summe von 1 bis 1 : 1

Generelles Vorgehen zur Summenberechnung

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, lös es direkt.
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Generelles Vorgehen zur Summenberechnung

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt.
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Summenberechnung:

- Die Summe von 1 bis 1 ist 1.
- Addiere n zur
 - Summe von 1 bis $n - 1$.

Generelles Vorgehen zur Summenberechnung

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt.
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Summenberechnung:

- Die Summe von 1 bis 1 ist 1.
- Addiere n zur
 - Summe von 1 bis $n - 1$.

Definition:

- $\text{sum}(1) = 1$
- $\text{sum}(n) = n + \text{sum}(n-1)$

Methodenaufrufe zur Summenberechnung

Definition:

- $\text{sum}(1) = 1$
- $\text{sum}(n) = n + \text{sum}(n-1)$

Aufrufe:

```
sum( 5 )
= 5 + sum( 4 )
= 5 + ( 4 + sum( 3 ) )
= 5 + ( 4 + ( 3 + sum( 2 ) ) )
= 5 + ( 4 + ( 3 + ( 2 + sum( 1 ) ) ) )
= 5 + ( 4 + ( 3 + ( 2 + 1 ) ) )
= 5 + ( 4 + ( 3 + 3 ) )
= 5 + ( 4 + 6 )
= 5 + 10
= 15
```

Quiz: square

Gegeben sei die folgende rekursive Definition:

- $\text{square}(1) = 1$
- $\text{square}(n) = \text{square}(n-1) + 2*n - 1$

Welches ist die korrekte Berechnung von `square(3)`?

- $\text{square}(3) = \text{square}(2) + \text{square}(1)$
- $\text{square}(3) = \text{square}(3) + \text{square}(2)$
- $\text{square}(3) = \text{square}(2) + 2*3 - 1$
- $\text{square}(3) = \text{square}(3) + 2*3 - 1$

Wie oft wird `square` insgesamt aufgerufen?

Quiz: square

Gegeben sei die folgende rekursive Definition:

- $\text{square}(1) = 1$
- $\text{square}(n) = \text{square}(n-1) + 2*n - 1$

Welches ist die korrekte Berechnung von `square(3)`?

- $\text{square}(3) = \text{square}(2) + \text{square}(1)$
- $\text{square}(3) = \text{square}(3) + \text{square}(2)$
- $\text{square}(3) = \text{square}(2) + 2*3 - 1$ (korrekt)
- $\text{square}(3) = \text{square}(3) + 2*3 - 1$

Wie oft wird `square` insgesamt aufgerufen? (3 Mal)

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

Generelles Vorgehen als Pseudocode

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt. (*Basisfall*)
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Generelles Vorgehen als Pseudocode

Eine rekursive Lösung besteht aus zwei Teilen:

- Wenn das Problem einfach ist, löse es direkt. (*Basisfall*)
- Andernfalls, teile es in einfachere Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.

Pseudocode:

```
int solution( int n ) {  
    if ( Basisfall ) {  
        return <etwas leicht berechenbares>;  
    }  
    else {  
        Teile das Problem in kleinere Teile;  
        return <etwas aus den Teillösungen berechnetes>;  
    }  
}
```

Summe der natürlichen Zahlen bis n

Definition:

- $\text{sum}(1) = 1$
- $\text{sum}(n) = n + \text{sum}(n-1)$

Summe der natürlichen Zahlen bis n

Definition:

- $\text{sum}(1) = 1$
- $\text{sum}(n) = n + \text{sum}(n-1)$

Java Code:

```
public int sum(int n) {  
    if (n == 1) {  
        return 1;  
    }  
    else {  
        return n + sum(n-1);  
    }  
}
```

Vergleich iterativ vs. rekursiv

Rekursiv:

```
public int sum(int n) {  
    if ( n == 1 ) {  
        return 1;  
    }  
    else {  
        return n + sum( n-1 );  
    }  
}
```

Iterativ:

```
public int sum(int n) {  
    int sum = 0;  
    for(int i=1; i<=n; i++){  
        sum += i;  
    }  
    return sum;  
}
```

Quiz: Fehlersuche

Gegeben sei die folgende rekursive Definition:

- $\text{square}(1) = 1$
- $\text{square}(n) = \text{square}(n-1) + 2*n - 1$

Welche Fehler sind in dem folgenden Code?

```
public int square(int n) {  
    n = 1;  
    return square(n-1) + 2*n - 1;  
}
```

Wie lautet der korrekte Code?

Quiz: Fehlersuche

Gegeben sei die folgende rekursive Definition:

- $\text{square}(1) = 1$
- $\text{square}(n) = \text{square}(n-1) + 2*n - 1$

Lösung:

```
public int square(int n) {  
    if ( n==1 ) {  
        return 1;  
    }  
    else {  
        return square(n-1) + 2*n - 1;  
    }  
}
```

Fibonaccizahlen

Definition:

- $\text{fibonacci}(0) = 0$
- $\text{fibonacci}(1) = 1$
- $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Fibonaccizahlen

Definition:

- fibonacci(0) = 0
- fibonacci(1) = 1
- fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

Java Code:

```
public int fibonacci(int n) {  
    if ( n == 0 ) {  
        return 0;  
    } else if ( n == 1 ) {  
        return 1;  
    } else {  
        return fibonacci( n-1 ) + fibonacci( n-2 );  
    }  
}
```

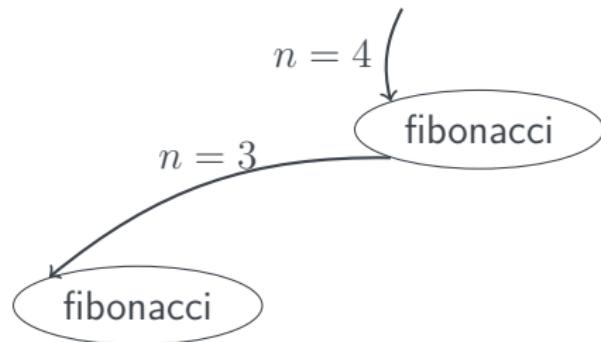
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



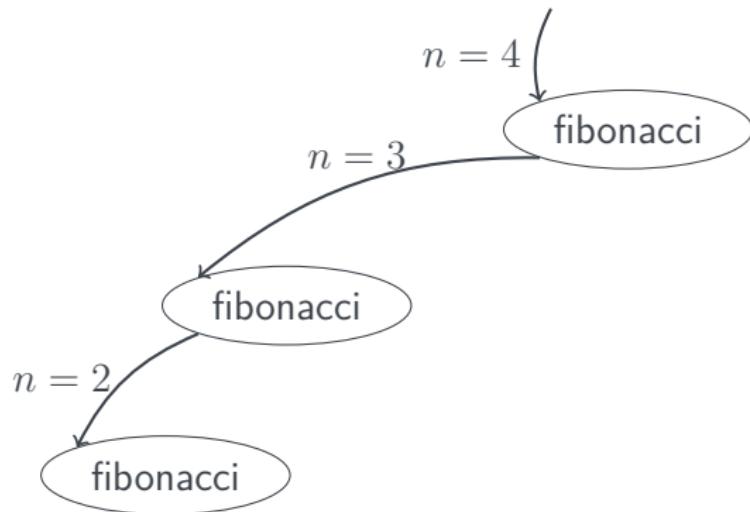
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



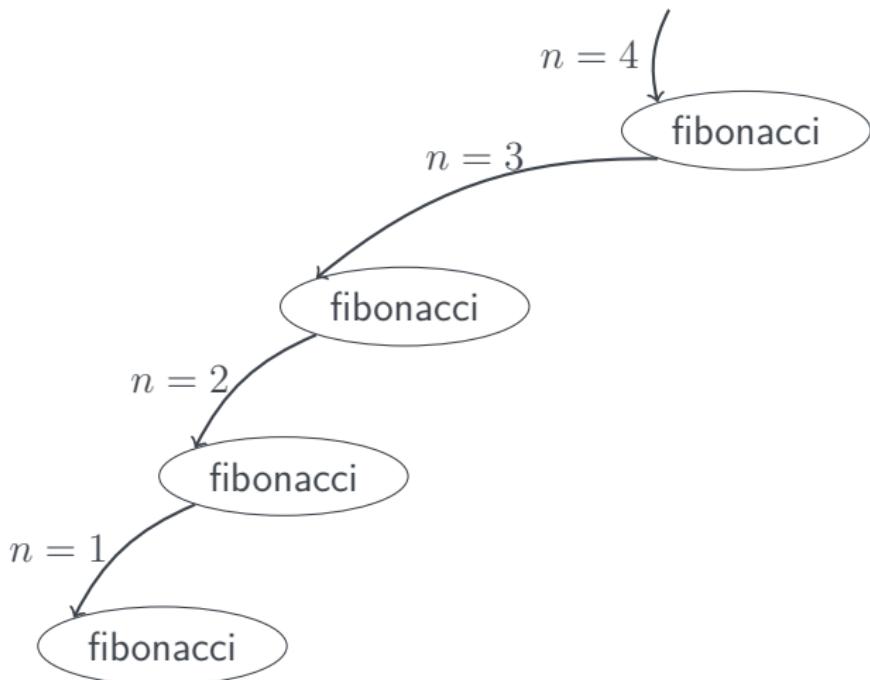
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



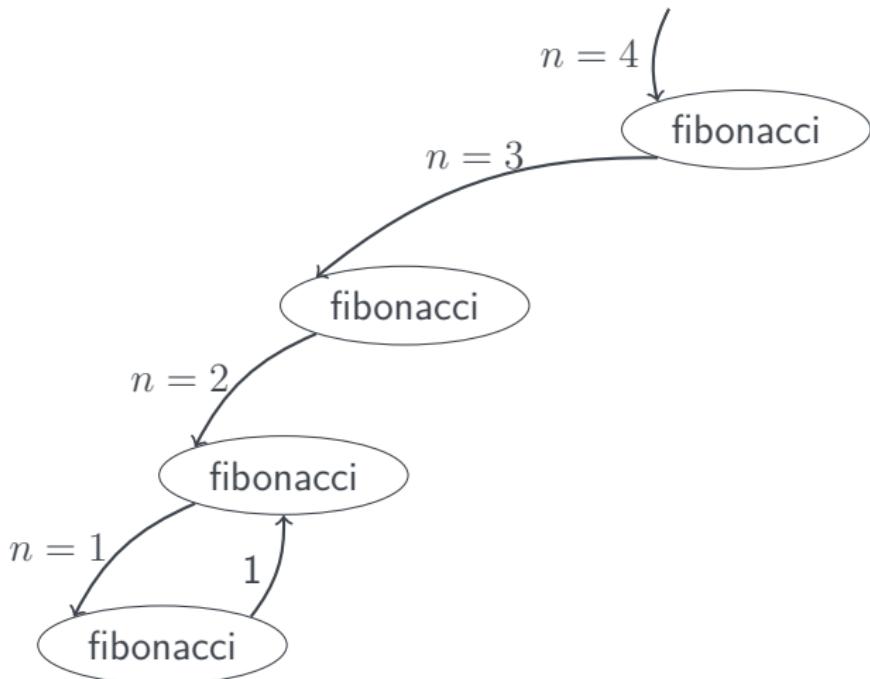
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



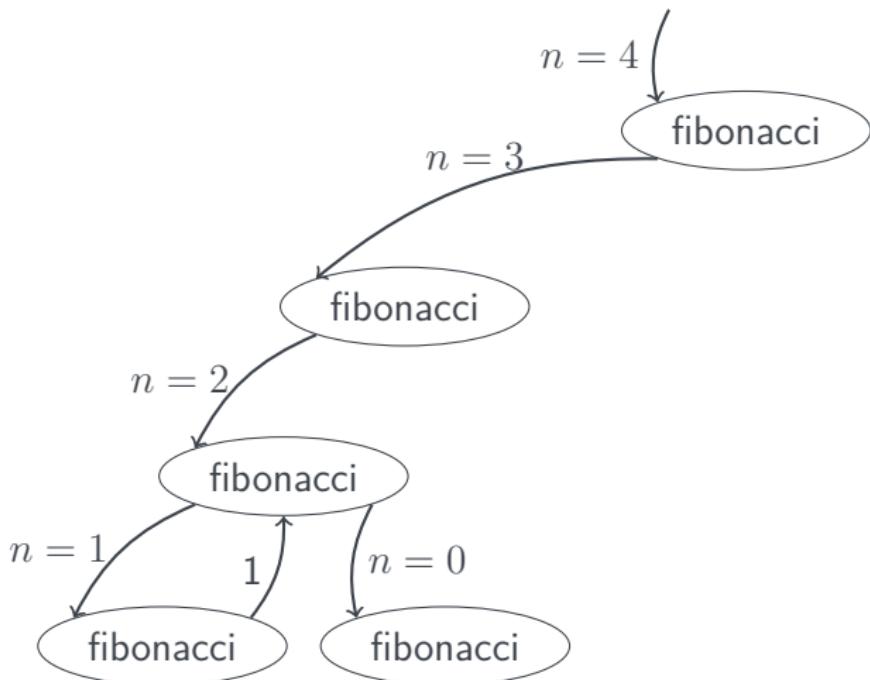
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



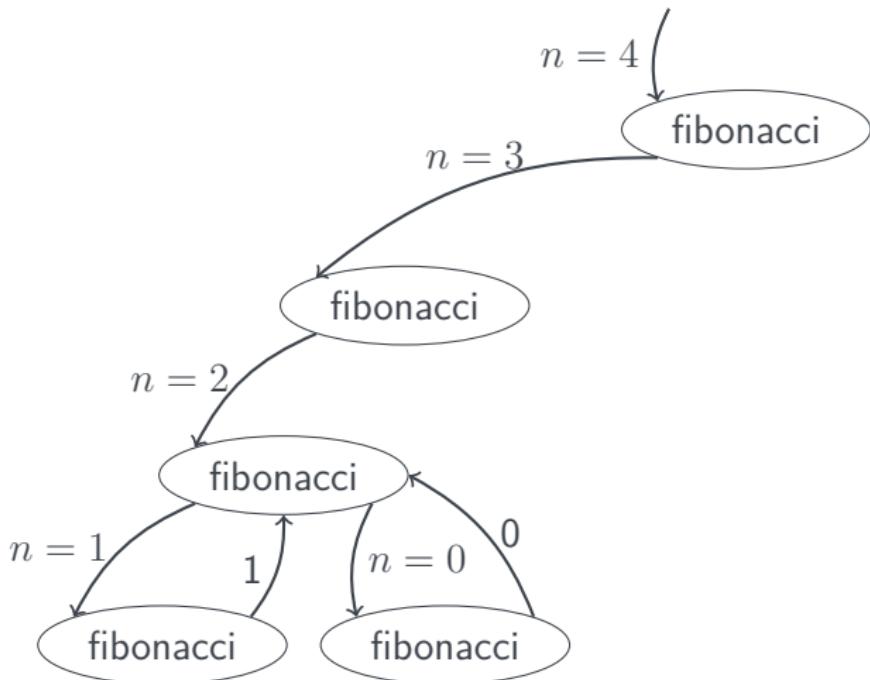
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



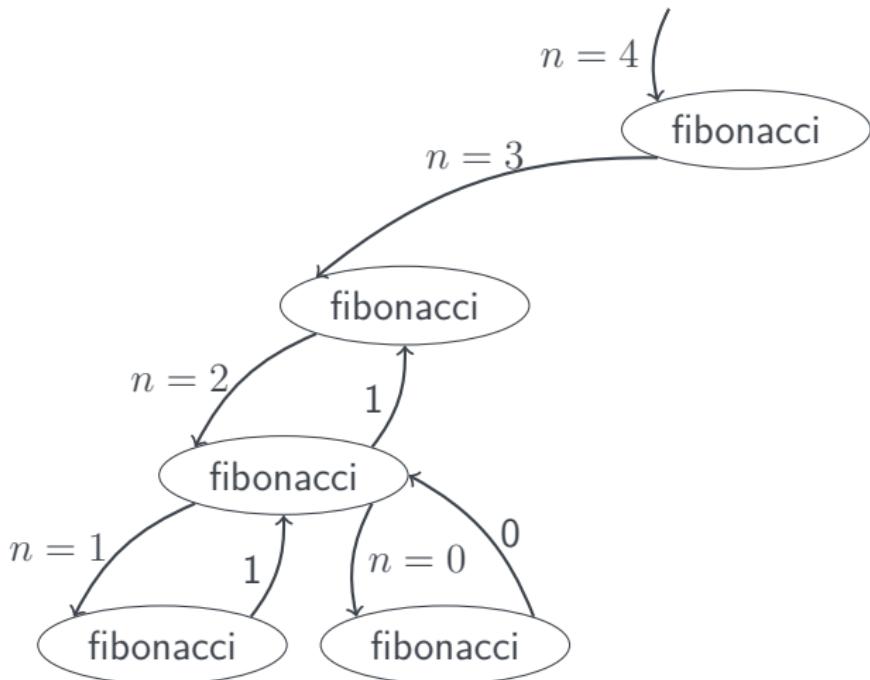
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



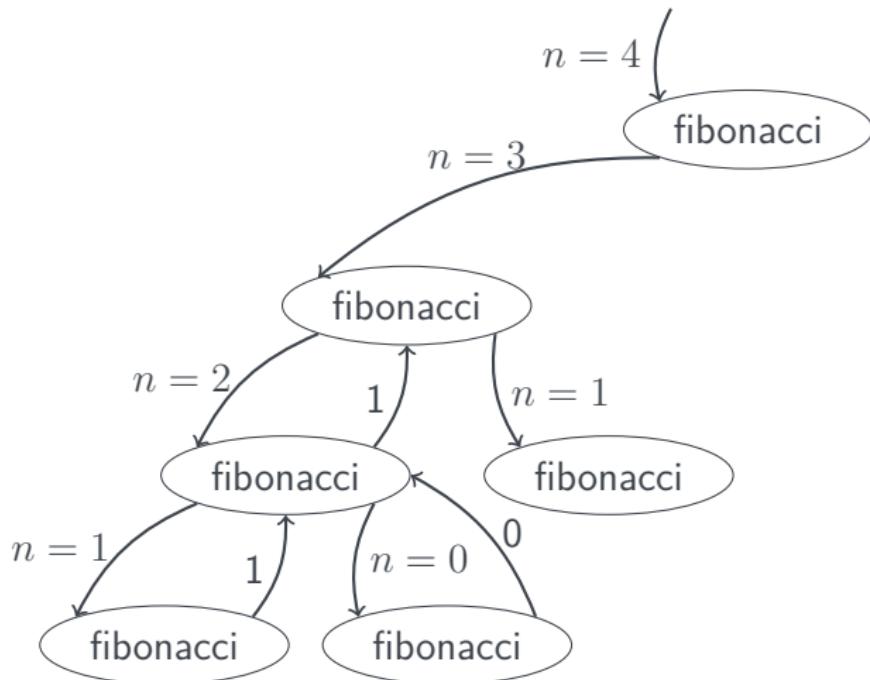
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



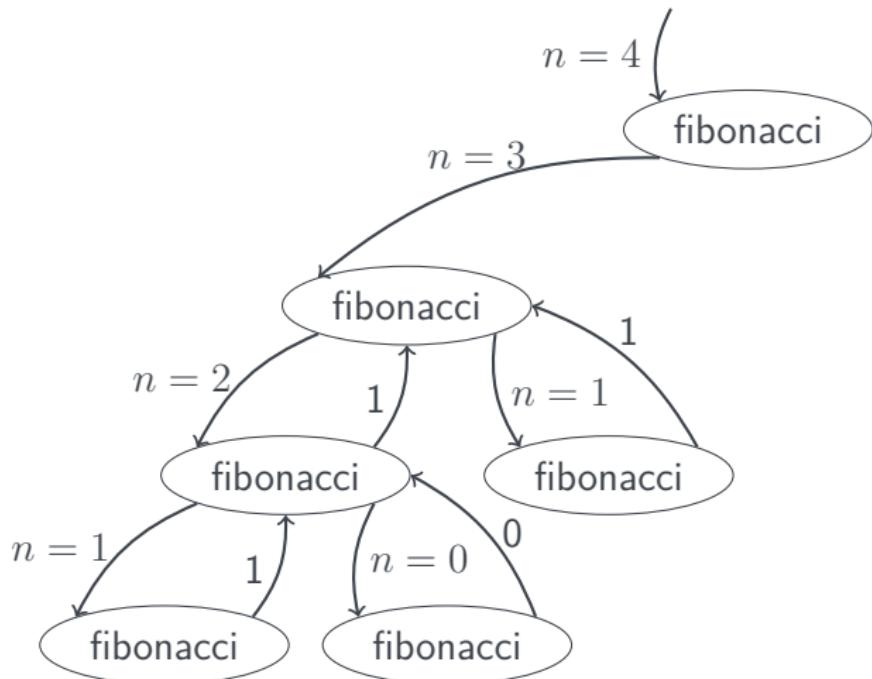
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



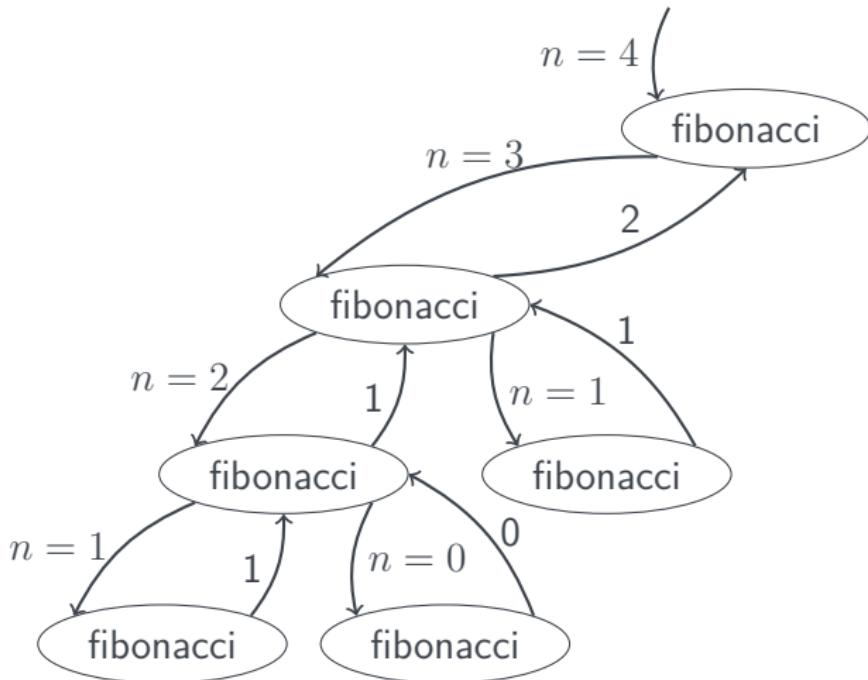
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



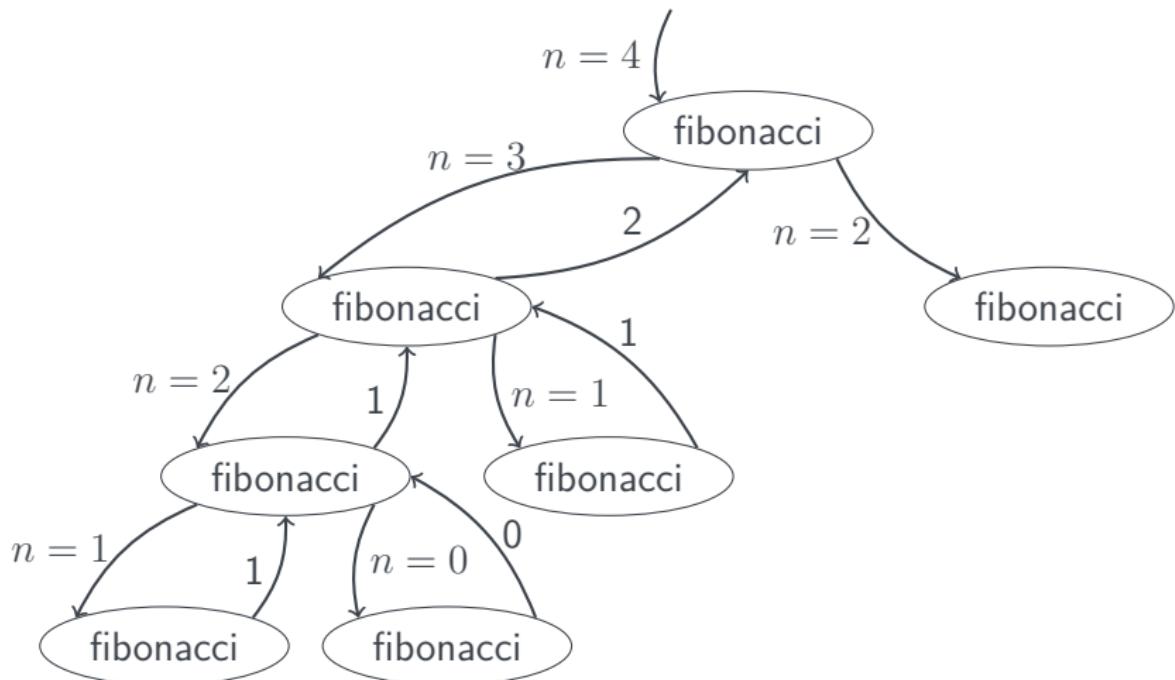
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



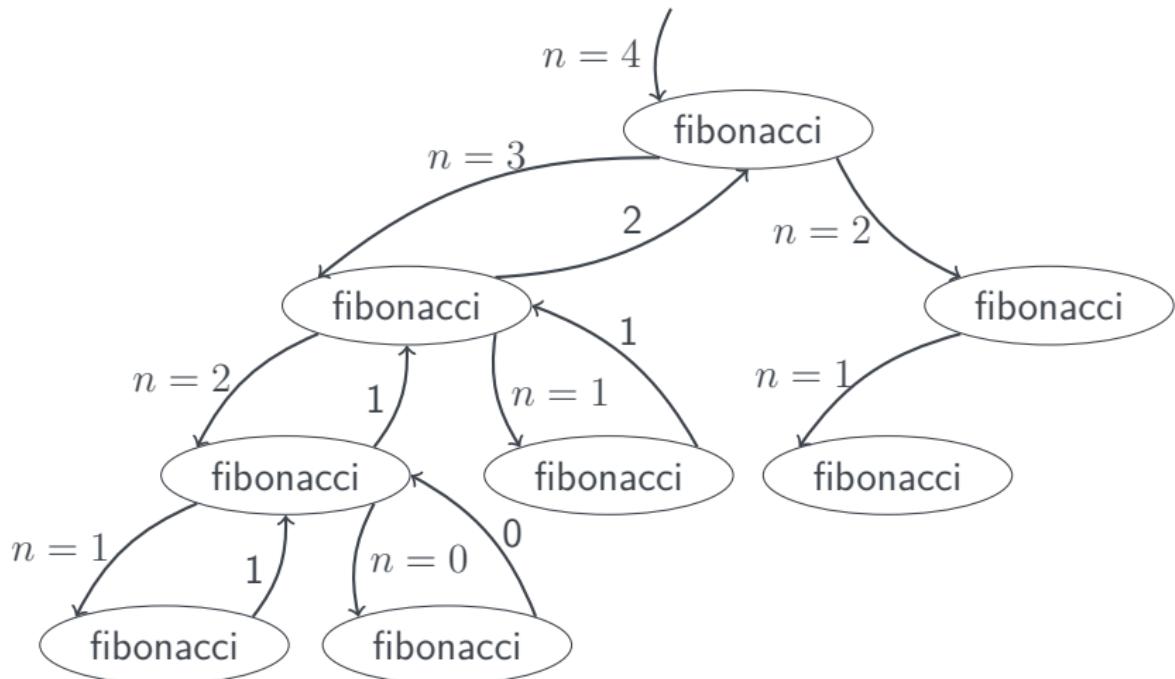
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



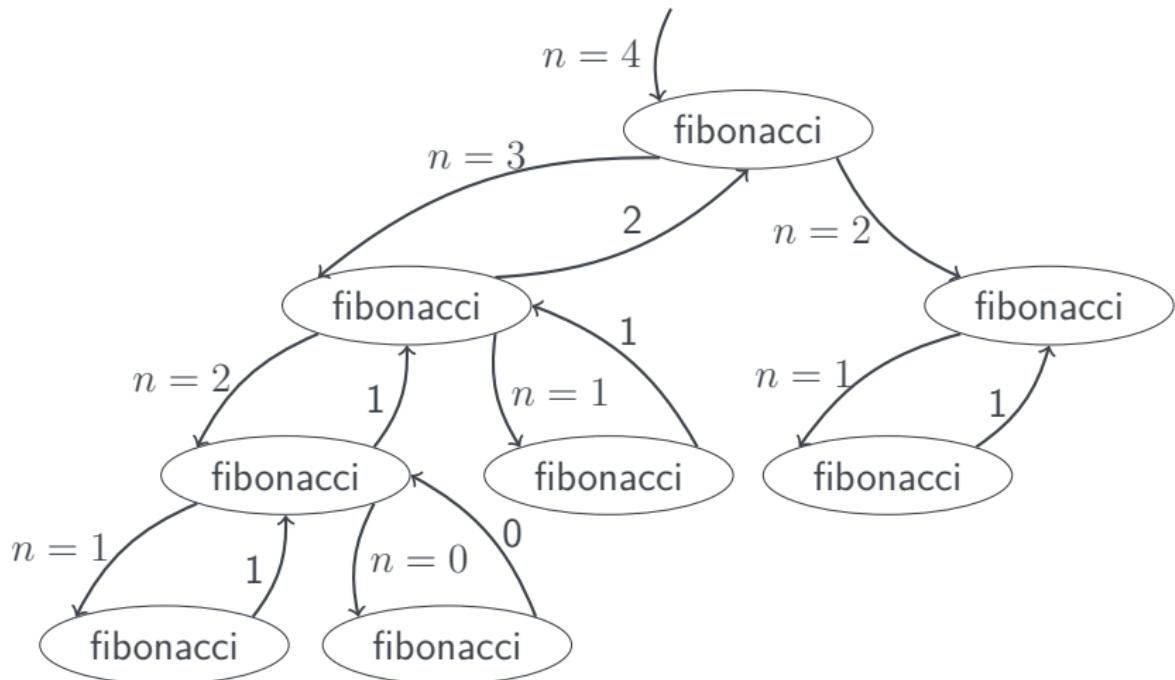
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



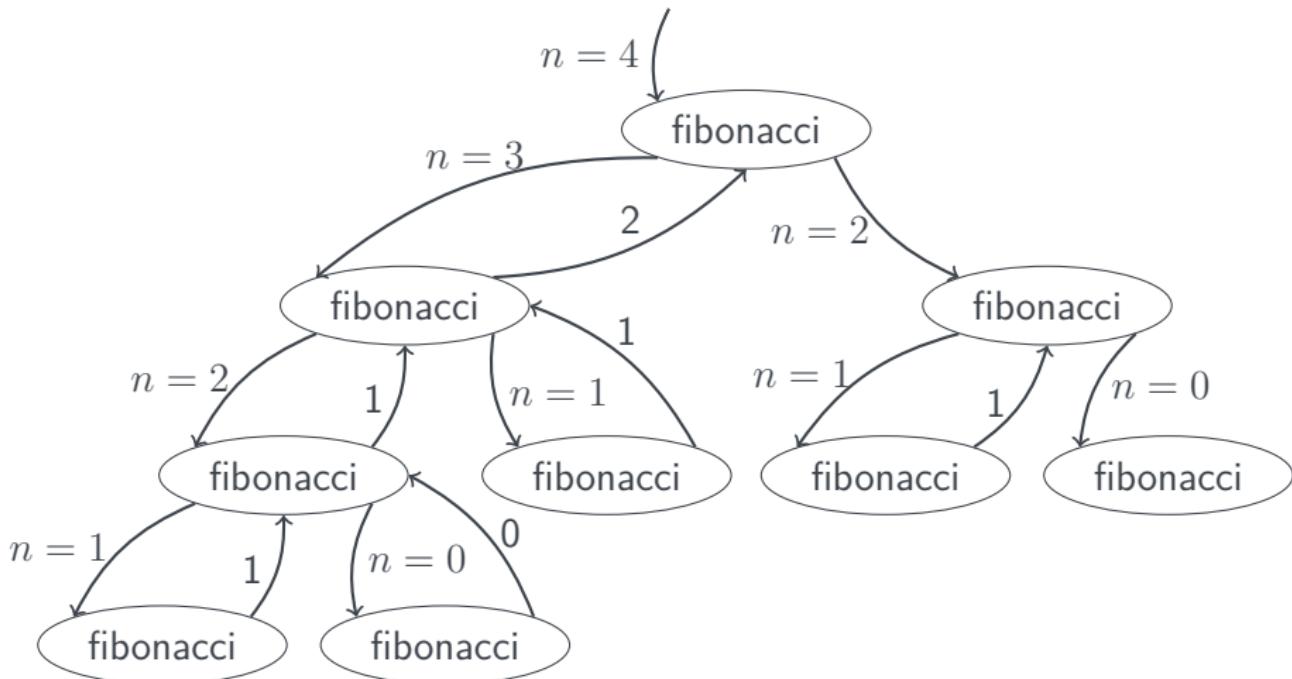
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



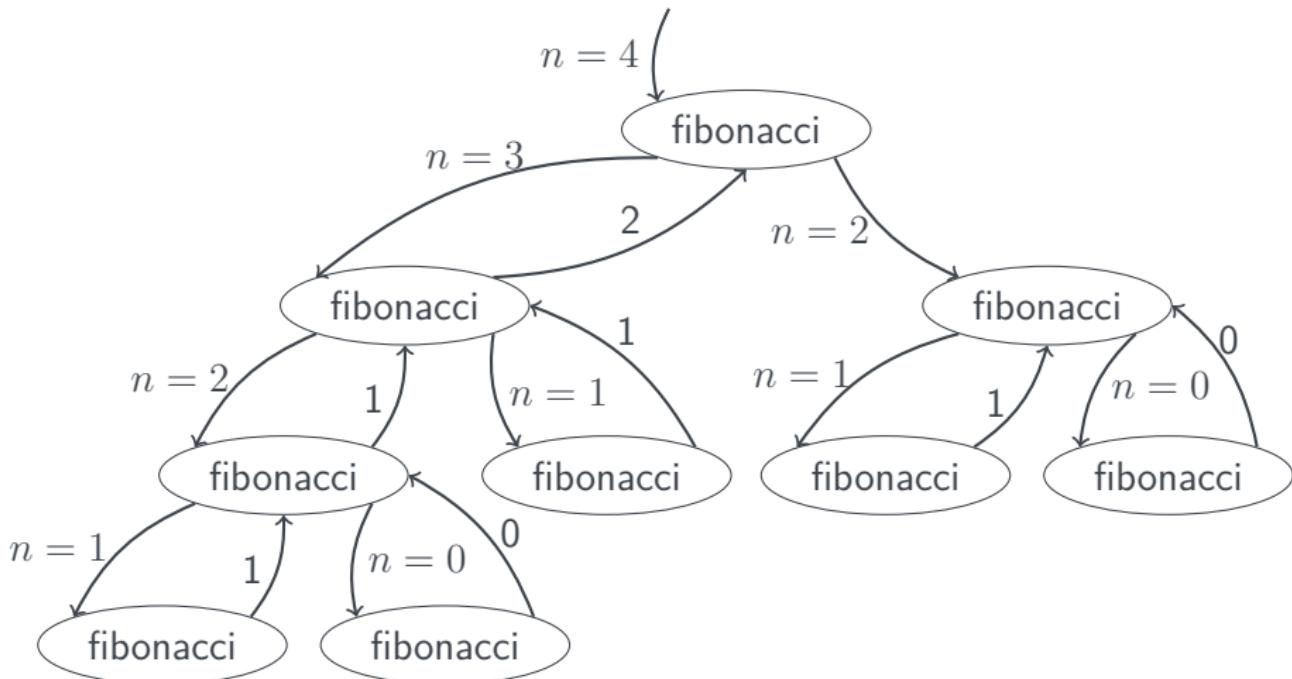
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



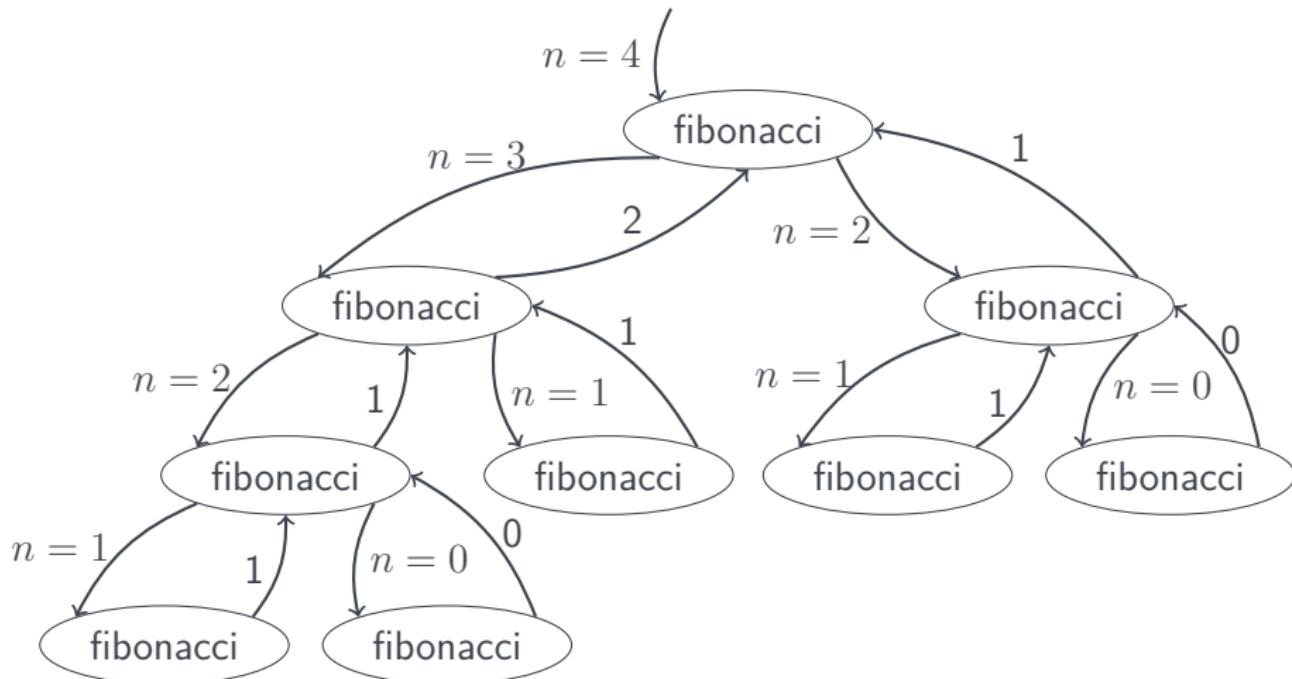
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



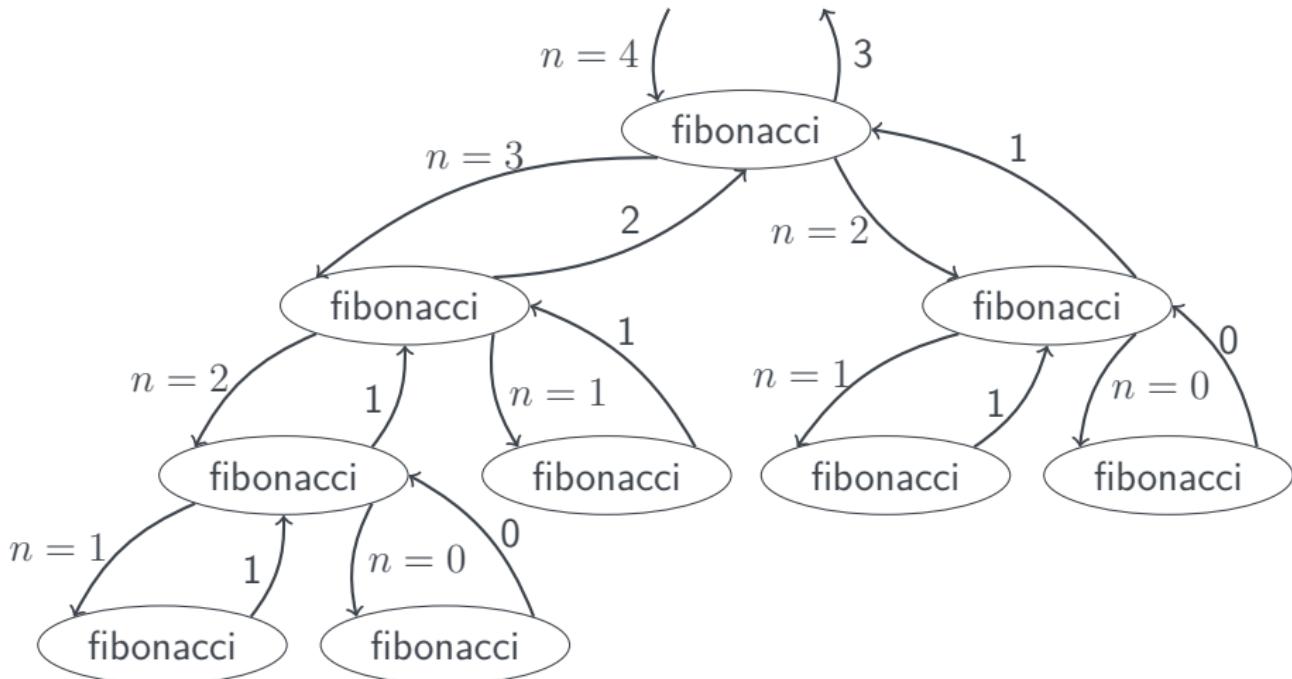
Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



Fibonaccizahlen Methodenaufrufe

```
fibonacci( n ) = fibonacci( n-1 ) + fibonacci( n-2 )
```



Quiz: Rekursion in Java

Gegeben sei die folgende rekursive Definition:

- $\text{div}(n, m) = 0$, für $n < m$.
- $\text{div}(n, m) = 1 + \text{div}(n-m, m)$, für $n \geq m$

Welche Aussagen sind korrekt?

- $\text{div}(7, 5)$ gibt 1 als Ergebnis.
- Bei $\text{div}(17, 5)$ wird `div` insgesamt 3 Mal aufgerufen.
- `div` kann auch iterativ berechnet werden.
- Die Implementierung von `div` lautet:

```
public int div( int n, int m ) {  
    if ( n < m ) { return 1;  
    } else { }  
    return 1 + divid( n-m, n );  
}
```

Quiz: Rekursion in Java

Gegeben sei die folgende rekursive Definition:

- $\text{div}(n, m) = 0$, für $n < m$.
- $\text{div}(n, m) = 1 + \text{div}(n-m, m)$, für $n \geq m$

Welche Aussagen sind korrekt?

- $\text{div}(7, 5)$ gibt 1 als Ergebnis. (korrekt)
- Bei $\text{div}(17, 5)$ wird `div` insgesamt 3 Mal aufgerufen. (korrekt)
- `div` kann auch iterativ berechnet werden. (korrekt)
- (falsch) Die folgende Implementierung von `div` ist korrekt:

```
public int div( int n, int m ) {  
    if ( n < m ) { return 0; }  
    else { return 1 + div( n-m, m ); }  
}
```

Outline

- 1 Exceptions
- 2 Rekursion
- 3 Rekursive Methodenaufrufe
- 4 Rekursion in Java
- 5 Zusammenfassung

Zusammenfassung: Exceptions

- Laufzeitfehler sind in Java Objekte der Klasse `Exception`.
- Code, der potentiell Fehler werfen könnte, kann in einen `try`-Block geschrieben werden. Darauf folgen ein oder mehrere `catch`-Blöcke, die die Fehler behandeln.
- Werden Exceptions nicht aufgefangen, werden sie weiter geworfen an die aufrufende Methode. Wird der Fehler nirgendwo behandelt, bricht die JVM die Ausführung des Programms ab.
- Es ist guter Stil, erwartete Fehler abzufangen und für den Benutzer sinnvolle (!) Fehlermeldungen zu geben.
- Exceptions sollten aber nicht verwendet werden, um normale Programmlogik zu ersetzen.

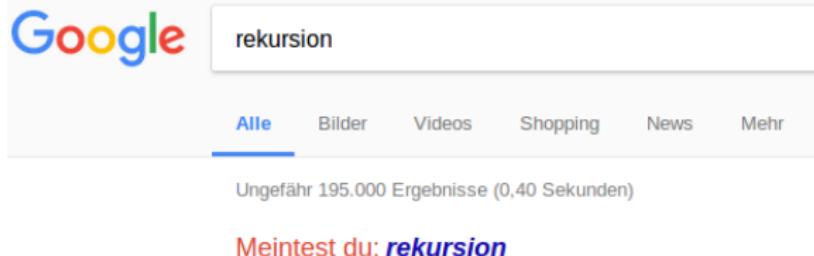
Zusammenfassung: Rekursion

- In der Informatik bedeutet Rekursion, dass eine Funktion in ihrer Definition **selbst** nochmals **aufgerufen** wird.
- Eine rekursive Lösung besteht aus zwei Teilen:
 - Wenn das Problem einfach ist, lös es direkt (**Basisfall**).
 - Andernfalls, teile es in **einfachere** Probleme und dann:
 - Löse die einfacheren Probleme auf dieselbe Art.
- Jedes iterativ lösbar Problem lässt sich auch rekursiv lösen.

```
void whatIsRecursion() {  
    if ( you understand Recursion ) {  
        System.out.println("yeah!");  
    }  
    else {  
        whatIsRecursion();  
    }  
}
```

Einschub: Rekursiver Humor

- Wörterbucheintrag für Rekursion: “Recursion, see Recursion.”
- “In order to understand recursion, one must first understand recursion” (auch als T-Shirt mit der Schrift im Kreis)
- Rekursive Acronyme, z. B.
 - GNU steht für “GNU is Not Unix”
 - PHP steht für “PHP Hypertext Preprocessor”
 - WINE steht für “WINE Is Not an Emulator”
 - TikZ steht für “TikZ ist kein Zeichenprogramm”
- Google Suche:





Vielen Dank!



Frank Schweiner

E-Mail frank.schweiner@mint-kolleg.de
Telefon +49 (0) 711 685-84326
Fax —

Universität Stuttgart
MINT-Kolleg Baden-Württemberg
Azenbergstr. 12
70174 Stuttgart