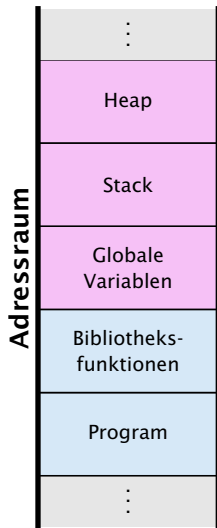


## 6 Speicherorganisation

Der Speicher des Programms ist in verschiedene Speicherbereiche untergliedert

- ▶ Speicherbereiche, die den eigentlichen Programmcode und den Code der Laufzeitbibliothek enthalten;
- ▶ einen Speicherbereich für **globale/statische Variablen**;
- ▶ einen Speicherbereich **Heap**, und
- ▶ einen Speicherbereich **Stack**.

Variablen werden üblicherweise auf dem Heap oder dem Stack gespeichert.



# Heap vs. Stack vs. statisch

## Heap

Auf dem Heap können zur Laufzeit zusammenhängende Speicherbereiche angefordert werden, und in beliebiger Reihenfolge wieder freigegeben werden.

## Stack

Der Stack ist ein Speicherbereich, auf dem neue Elemente oben gespeichert werden, und Freigaben in umgekehrter Reihenfolge (d.h. oben zuerst) erfolgen müssen (**LIFO** = **L**ast **I**n **F**irst **O**ut).

## Statische Variablen

Statische Variablen werden zu Beginn des Programms angelegt, und zum Ende des Programms wieder gelöscht.

In **Java** müssen Elemente auf dem Heap nicht explizit wieder freigegeben werden. Diese Freigabe übernimmt der **Garbage Collector**.

# Statische Variablen

**Statische Variablen** (auch Klassenvariablen) werden im Klassenrumpf **ausserhalb** einer Funktion mit dem zusätzlichen Schlüsselwort **static** definiert.

Jede Funktion der Klasse kann dann diese Variablen benutzen; deshalb werden sie manchmal auch globale Variablen genannt.

## Beispiel – Statische Variablen

```
1 public class GGT extends MiniJava {
2     static int x, y;
3     static void readInput() {
4         x = read();
5         y = read();
6     }
7     public static void main (String[] args) {
8         readInput();
9         while (x != y) {
10            if (x < y)
11                y = y - x;
12            else
13                x = x - y;
14        }
15        write(x);
16    }
17 }
```

# Verwendung des Heaps

Speicherallokation mit dem Operator `new`:

```
int[][] arr;  
arr = new int[10][]; // array mit int-Verweisen
```

Immer wenn etwas mit `new` angelegt wird, landet es auf dem Heap.

Wenn keine Referenz mehr auf den angeforderten Speicher existiert **kann** der **Garbage Collector** den Speicher freigeben:

```
int[][] arr;  
arr = new int[10][]; // array mit int-Verweisen  
arr = null; // jetzt koennte GC freigeben
```

# Verwendung des Heaps

## Beispiel:

```
1 public static int[] readArray(int number) {
2     // number = Anzahl zu lesender Elemente
3     int[] result = new int[number];
4     for (int i = 0; i < number; ++i) {
5         result[i] = read();
6     }
7     return result;
8 }
9 public static void main(String[] args) {
10     readArray(6);
11 }
```

Da die von `readArray` zurückgegebene Referenz nicht benutzt wird, kann der GC freigeben.

# Verwendung des Heaps

## Beispiel:

```
1 public static void main(String[] args) {  
2     int[] b = readArray(6);  
3     int[] c = b;  
4     b = null;  
5 }
```

Da `c` immer noch eine Referenz auf das array enthält erfolgt keine Freigabe.

# Verwendung des Stacks

- ▶ Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als **Frames** auf dem Stack gespeichert.
- ▶ Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- ▶ Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.



# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {
    z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Stack

Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {  
    z = 1;  
}  
  
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```



Stack

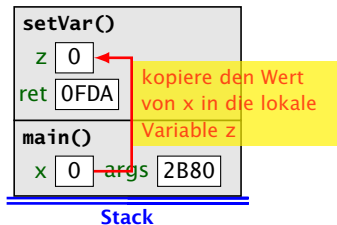
Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {  
    z = 1;  
}
```

```
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```



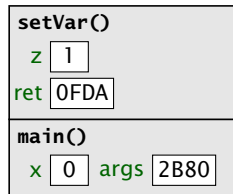
Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {  
    z = 1;  
}
```

```
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```



Stack

Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Value

Die Variable, die wir bei dem Aufruf übergeben, verändert ihren Wert nicht.

```
public static void setVar(int z) {  
    z = 1;  
}  
  
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
    write("x == " + x);  
}
```

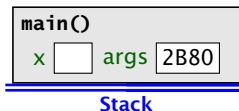


Stack

Das ist die einzige Form der Parameterübergabe, die **Java** unterstützt.

# Parameterübergabe – Call-by-Reference

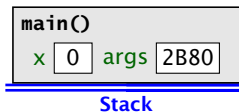
```
public static void setVar(int &z) {  
    z = 1;  
}  
  
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```



Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.

# Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {  
    z = 1;  
}  
  
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```

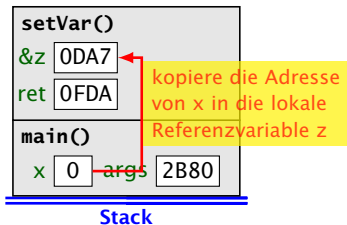


Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.

# Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {  
    z = 1;  
}
```

```
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```



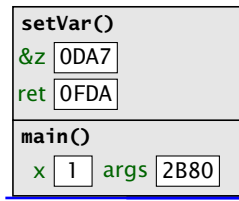
Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.



# Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {  
    z = 1;  
}
```

```
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
  
    write("x == " + x);  
}
```

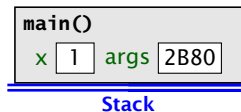


Stack

Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.

# Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {  
    z = 1;  
}  
  
public static void main(String[] args) {  
    int x;  
    x = 0;  
    setVar(x);  
    write("x == " + x);  
}
```



Diese Form der Parameterübergabe ist in **Java** nicht möglich, aber z.B. in **C++**.

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

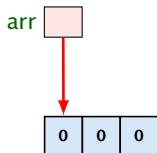
```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

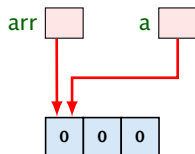


Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

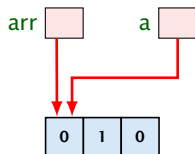


Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

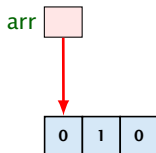


Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den **Inhalt** des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



Ausgabe: **arr[1] == 1**

## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a    = new int[3];  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

Ausgabe: `arr[1] == 0`

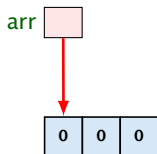


## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a    = new int[3];  
    a[1] = 1;  
}
```

```
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

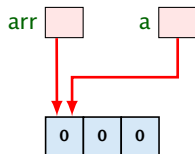


Ausgabe: `arr[1] == 0`

## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a    = new int[3];  
    a[1] = 1;  
}  
  
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



Ausgabe: `arr[1] == 0`

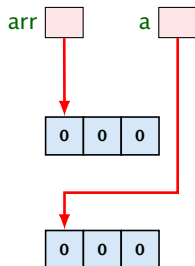
## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a = new int[3];  
    a[1] = 1;  
}
```

```
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```

Ausgabe: `arr[1] == 0`

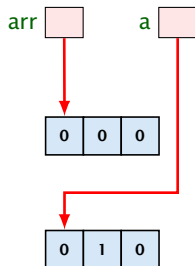


## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a    = new int[3];  
    a[1] = 1;  
}
```

```
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



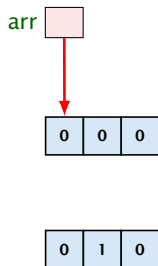
Ausgabe: `arr[1] == 0`

## Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {  
    a    = new int[3];  
    a[1] = 1;  
}
```

```
public static void main(String[] args) {  
    // initialize array elements to 0  
    int[] arr = new int[3];  
    setVar(arr);  
    write("arr[1] == " + arr[1]);  
}
```



Ausgabe: `arr[1] == 0`

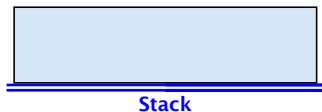
# Rekursive Funktionen

**Rekursive Funktionen** sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



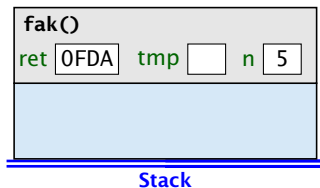
# Rekursive Funktionen

**Rekursive Funktionen** sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



# Rekursive Funktionen

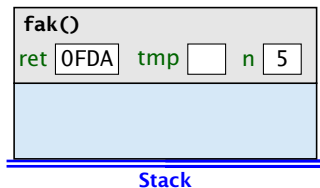
**Rekursive Funktionen** sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```





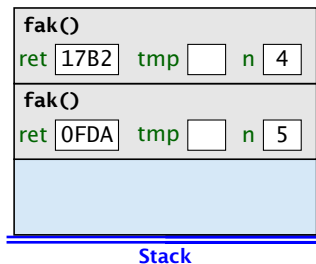
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



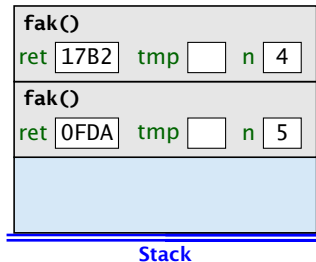
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



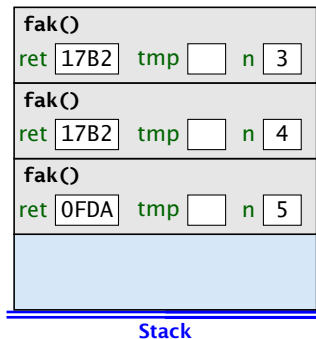
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



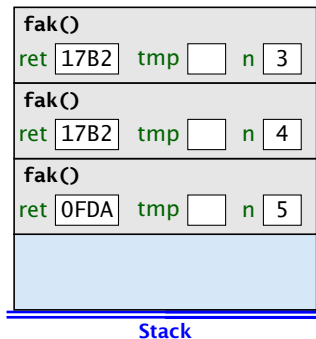
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



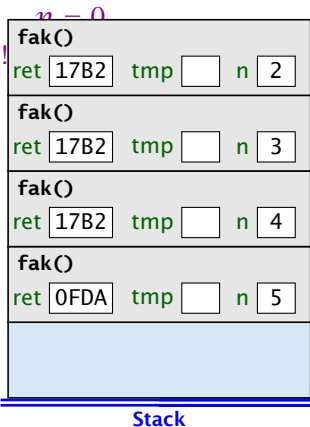
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



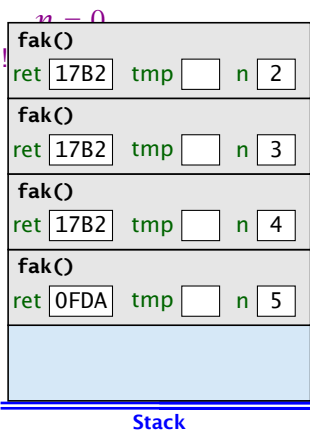
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



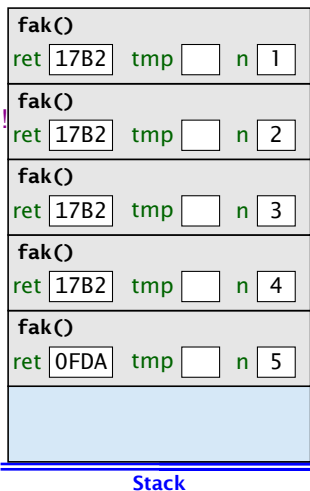
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



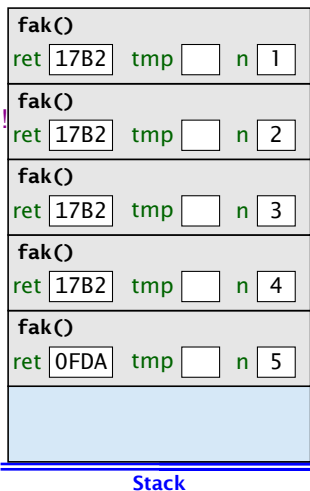
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```





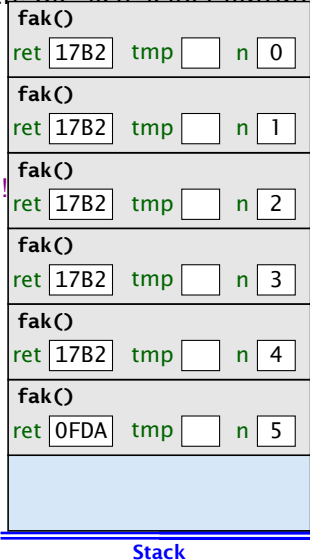
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



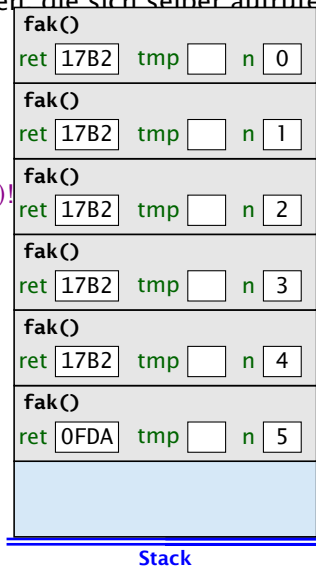
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



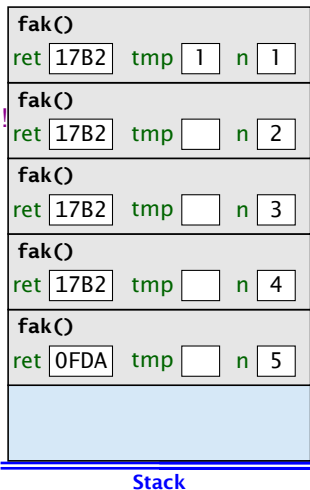
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



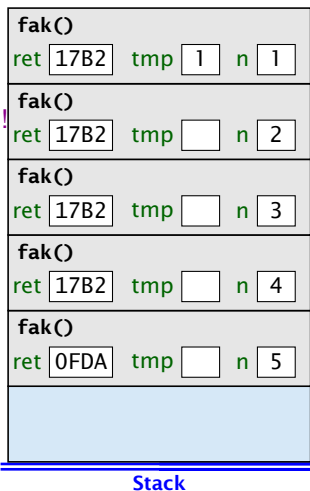
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



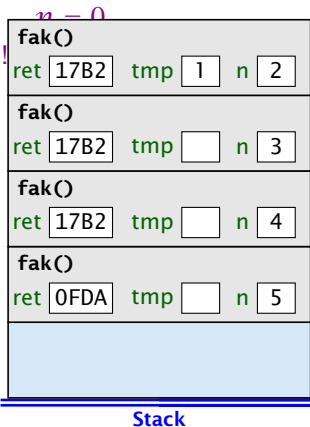
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



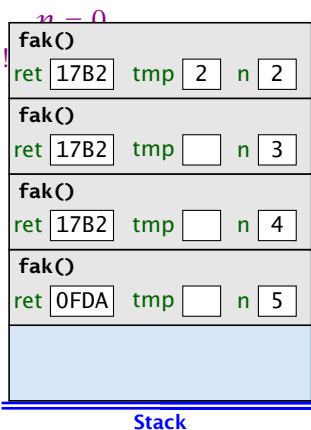
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



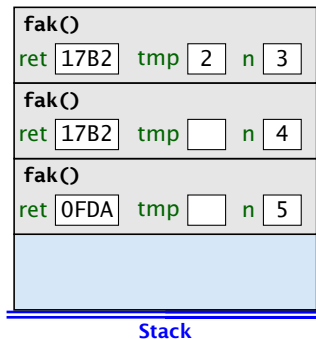
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



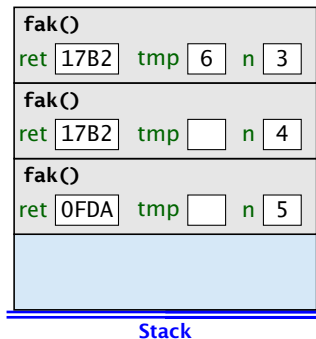
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```





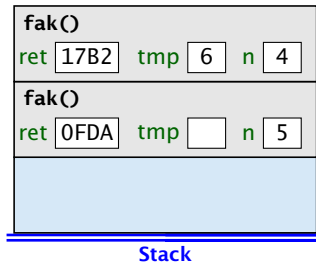
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



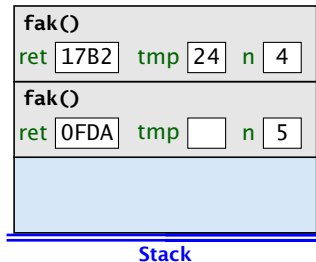
# Rekursive Funktionen

Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {  
    long tmp;  
  
    if (n > 0) {  
        tmp = fak(n-1);  
        tmp *= n;  
        return tmp;  
    }  
    else return 1;  
}
```



# Rekursive Funktionen

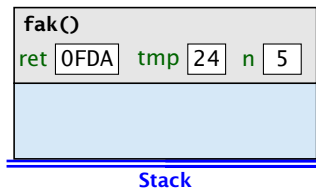
**Rekursive Funktionen** sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```



# Rekursive Funktionen

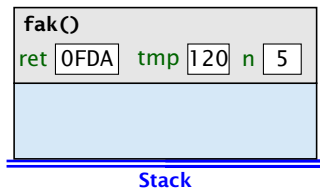
**Rekursive Funktionen** sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

**Beispiel:** Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

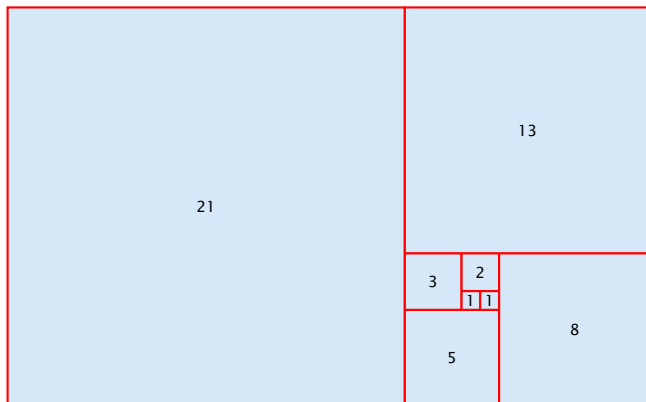


# Vollständiger Code

```
1 public class Fakultae {
2     public static long fak(int n) {
3         if (n > 0)
4             return n * fak(n-1);
5         else
6             return 1;
7     }
8     public static void main(String args[]) {
9         System.out.println(fak(20));
10    }
11 }
```

# Fibonaccizahlen

$$F_n = \begin{cases} n & 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$



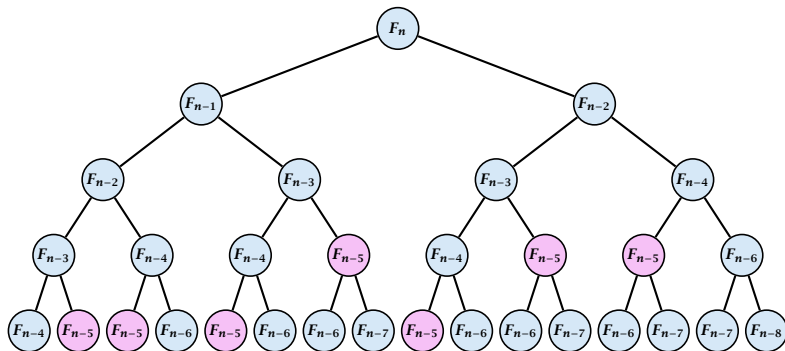
# Vollständiger Code

```
1 public class Fibonacci {
2     public static long fib(int n) {
3         if (n > 1)
4             return fib(n-1)+fib(n-2);
5         else
6             return n;
7     }
8
9     public static void main(String args[]) {
10        System.out.println(fib(50));
11    }
12 }
```

# Fibonaccizahlen

Programmmlauf benötigt mehr als 1 min.

Warum ist das so langsam?



Wir erzeugen viele rekursive Aufrufe für die gleichen Teilprobleme!



## Lösung

- ▶ Speichere die Lösung für ein Teilproblem in einer **globalen Variable**.
- ▶ Wenn das Teilproblem das nächste mal gelöst werden soll braucht man nur nachzuschauen...

# Vollständiger Code

```
1 public class FibonacciImproved {
2     // F93 does not fit into a long
3     static long[] lookup = new long[93];
4
5     public static long fib(int n) {
6         if (lookup[n] > 0) return lookup[n];
7
8         if (n > 1) {
9             lookup[n] = fib(n-1)+fib(n-2);
10            return lookup[n];
11        } else
12            return n;
13    }
14    public static void main(String args[]) {
15        System.out.println(fib(50));
16    }
17 }
```

Hier nutzen wir die Tatsache, dass der `new`-Operator `long`-Variablen mit dem Wert Null initialisiert.

**Achtung:** lokale Variablen werden nicht initialisiert.