

Eprog Theorie Teil

Aaron Zeller

December 13, 2023

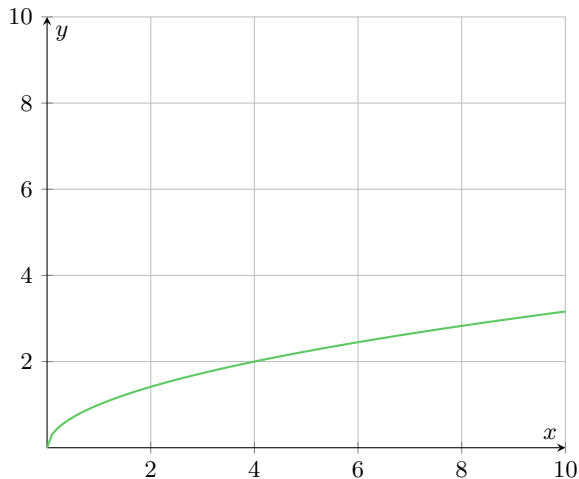


Exercise Types

- Expression Evaluation
- Code Execution
- Loop Invariante
- Loop Invariante + Precondition
- Klassenrätsel I
- Klassenrätsel II
- Klassenrätsel III
- EBNF G/U
- Weakest Precondition
- Redundanter Code
- Immer, Manchmal, Nie
- EBNF Äquivalente Regeln
- Hoare Triple
- Das kommt nie an der Prüfung

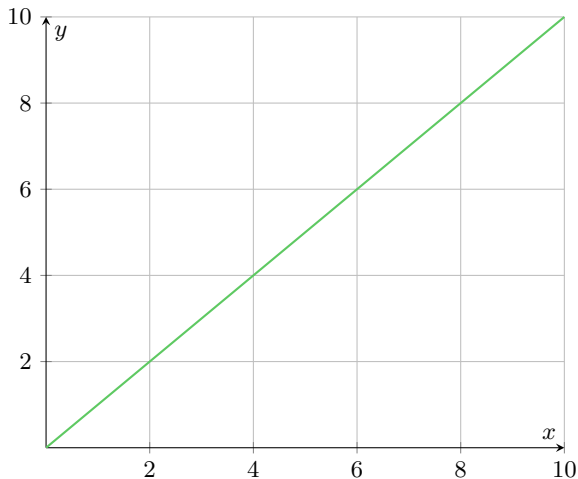
Learning Curve

- Flattening Learning Curve (1) (2)
- Linear Learning Curve
- Polynomial Learning Curve



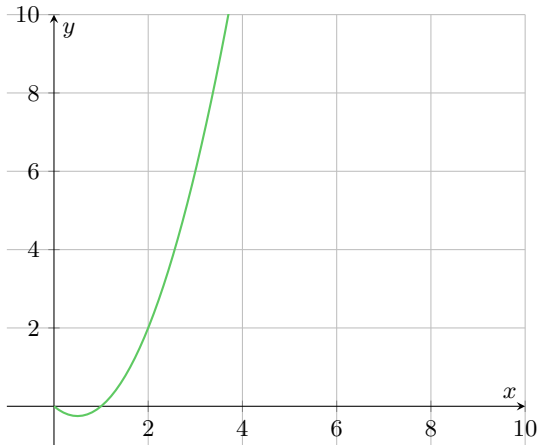
Learning Curve

- Flattening Learning Curve
- **Linear Learning Curve (3)**
- Polynomial Learning Curve



Learning Curve

- Flattening Learning Curve
- Linear Learning Curve
- Polynomial Learning Curve (4) (5)

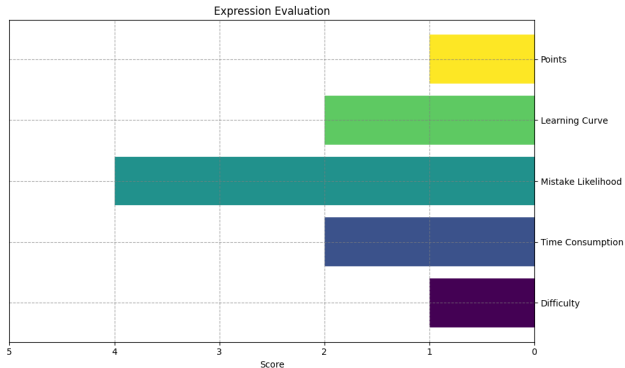


Expression Evaluation

```
1 public class Example {  
2     public static void main(String[] args) {  
3         System.out.println(3 + 2 - 5 % 3 / 1 - (2 + 1))  
4     }  
5 }
```

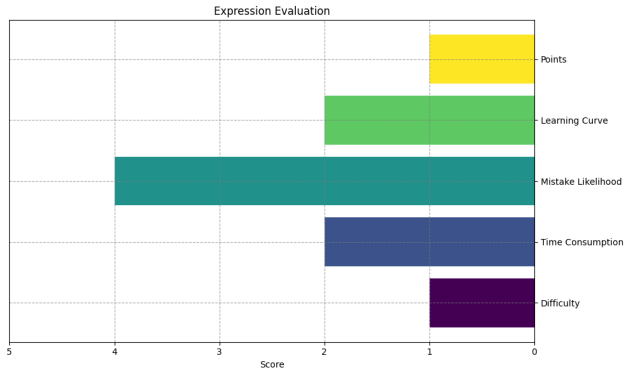
Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



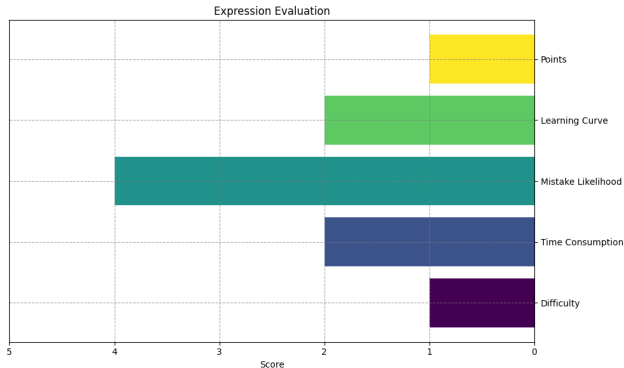
Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



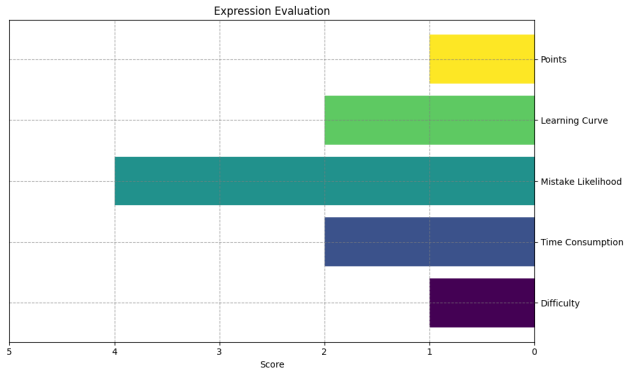
Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



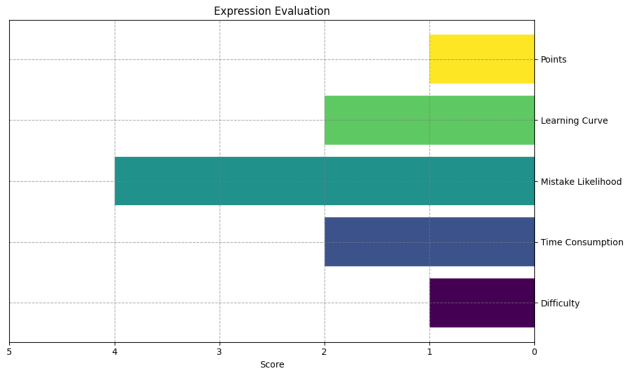
Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



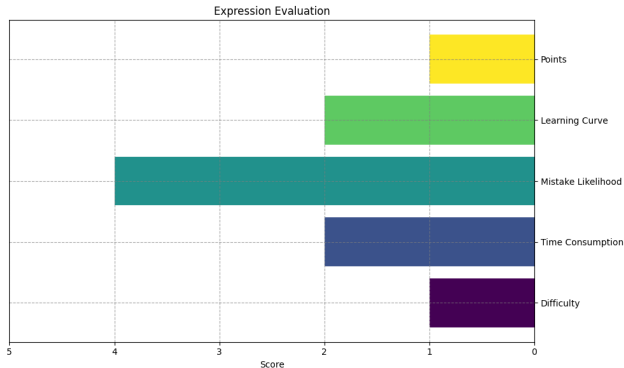
Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



Expression Evaluation

- Punkt (*, /, %) vor Strich (-, +)
- Punkt (&&) vor Strich (||)
- Rechtsassoziativ vs Linksassoziativ
- + für Strings vs. + für int, long, double, etc.
- Benutzt JShell!



Expression Evaluation

Operators	Precedence
postfix increment and decrement	<code>++ -</code>
prefix increment and decrement, and unary	<code>++ - + - ~!</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>
shift	<code><< >> >>></code>
relational	<code>< > <= >= instanceof</code>
equality	<code>== !=</code>
bitwise AND	<code>&</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&&</code>
logical OR	<code> </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &= ^= =</code>

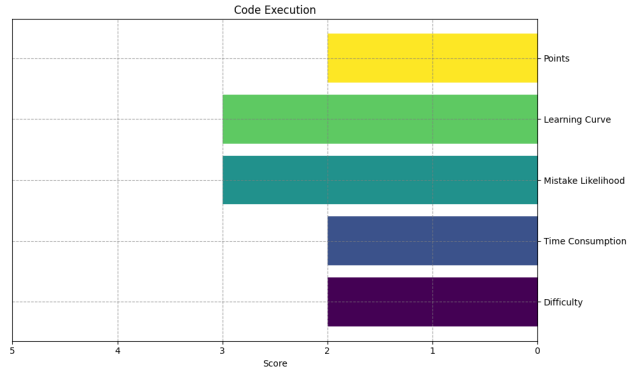
Table: Java Operator Precedence

Code Execution

```
1  public class Example {  
2      public void foo(int x) { x = 2; }  
3      public void foo(int[] x) { x[0] = 2; }  
4  
5      public static void main(String[] args) {  
6          int xInt = 0;  
7          int[] xArr = new int[] {0};  
8  
9          foo(xInt);  
10         foo(xArr);  
11  
12         System.out.println(xInt);    // 1  
13         System.out.println(xArr);    // 2  
14     }  
15 }
```

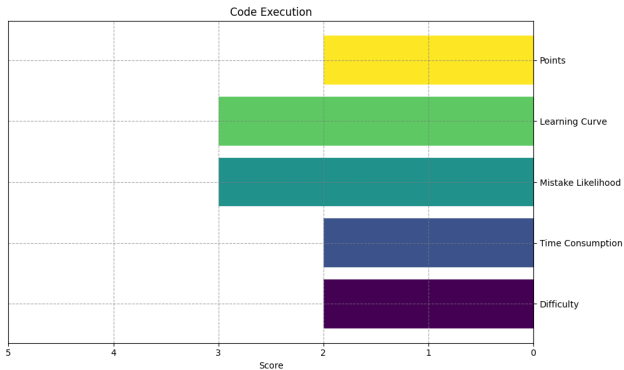
Code Execution

- Reference semantics vs Value semantics
- Rekursive Funktionsaufrufe
- Method Overloading
- Loop Execution mit Sprüngen



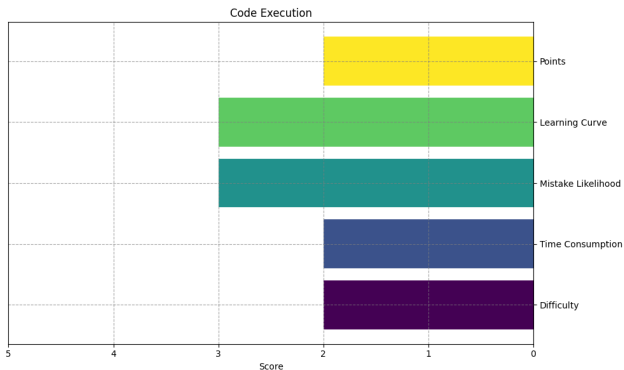
Code Execution

- Reference semantics vs Value semantics
 - Rekursive Funktionsaufrufe
 - Method Overloading
 - Loop Execution mit Sprüngen



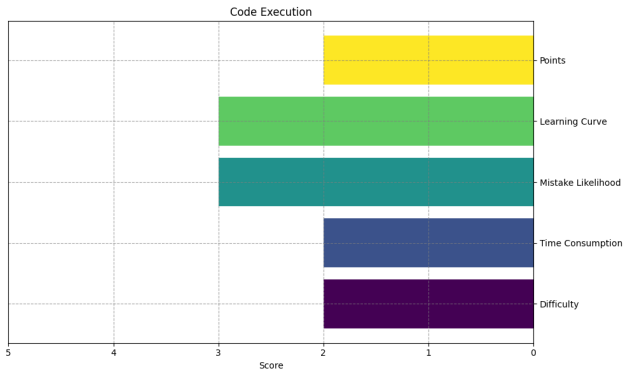
Code Execution

- Reference semantics vs Value semantics
- Rekursive Funktionsaufrufe
- Method Overloading
- Loop Execution mit Sprüngen



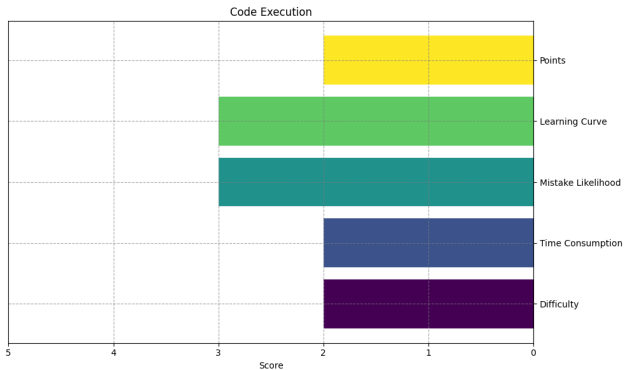
Code Execution

- Reference semantics vs Value semantics
- Rekursive Funktionsaufrufe
- Method Overloading
- Loop Execution mit Sprüngen



Code Execution

- Reference semantics vs Value semantics
- Rekursive Funktionsaufrufe
- Method Overloading
- Loop Execution mit Sprüngen

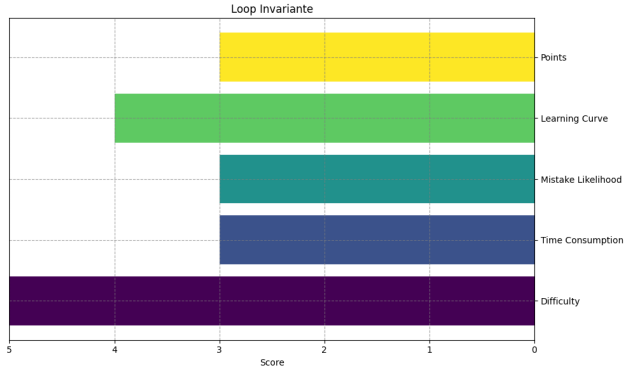


Loop Invariante

```
1  public int compute(int n) {
2      // Precondition: n >= 0
3      int x;
4      int res;
5      x = 0;
6      res = x;
7      // Loop Invariante:
8      while (x <= n) {
9          res = res + x;
10         x = x + 1;
11     }
12     // Postcondition: res == ((n + 1) * n) / 2
13     return res;
14 }
```

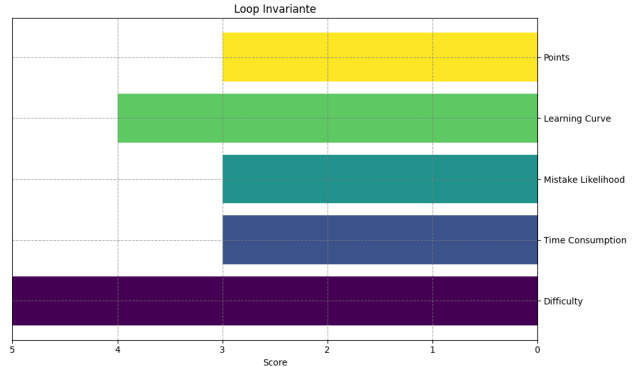
Loop Invariante

- Benutzt das Rezept auf eprog.ch 
- Loop Invarianten brauchen viel Übung.
- Kommen immer mehr an der Prüfung.




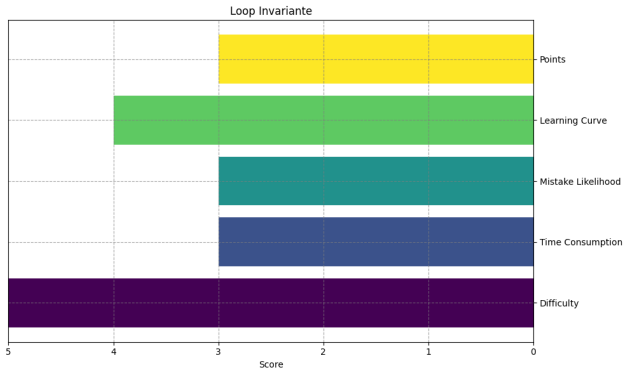
Loop Invariante

- Benutzt das Rezept auf eprog.ch 
- Loop Invarianten brauchen viel Übung.
- Kommen immer mehr an der Prüfung.




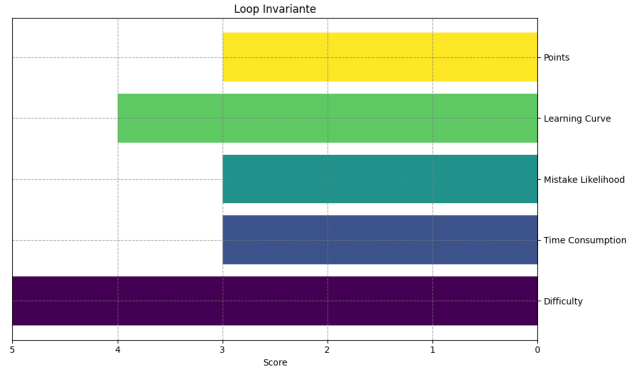
Loop Invariante

- Benutzt das Rezept auf eprog.ch 
- Loop Invarianten brauchen viel Übung.
- Kommen immer mehr an der Prüfung.



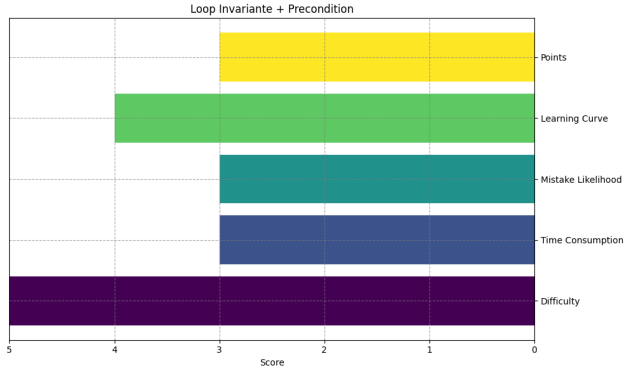
Loop Invariante

- Benutzt das Rezept auf eprog.ch 
- Loop Invarianten brauchen viel Übung.
- Kommen immer mehr an der Prüfung.



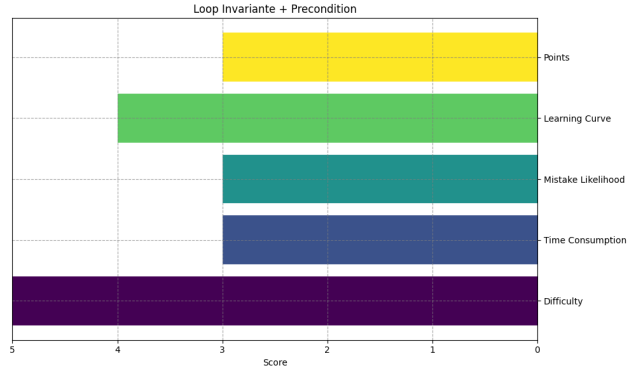
Loop Invariante + Precondition

- HS20 Prüfung + Implizit in FS22
- Gleiches Muster - Precondition so wählen, dass Postcondition folgt.
- Trial & Error.



Loop Invariante + Precondition

- HS20 Prüfung + Implizit in FS22
- Gleiches Muster - Precondition so wählen, dass Postcondition folgt
- Trial & Error.



Loop Invariante + Precondition - HS20

```
1  static int arraySum(int[] iArray) {
2      if (iArray.length == 0) {
3          throw new IllegalArgumentException();
4      }
5      int k = 1;
6      int result = iArray[0];
7      // P (precondition): // I (invariante):
8      while (k != iArray.length) {
9          result += iArray[k];
10         k++;
11     }
12     // Q (postcondition): result = sum(iArray, 0, N-1)
13     // N = Anzahl Elemente in iArray
14     return result;
15 }
```

Loop Invariante + Precondition (Implizit) - FS22

Raphaël Larisch @rlarisch · 11 months ago · edited 11 months ago





0

+

-

19

+




v == 2 is important, if this is missing we can't be sure that we have calculated 
end is necessary, but it is not wrong I would say.

+ Add Comment

... More

Simon Sure @sisure · 10 months ago · edited 10 months ago



I understand why $v == 2$ must hold for us to  But the loop invariant must also hold before the loop, and we have no knowledge whether  Maybe it would have had to be specified as a precondition as part of the task...

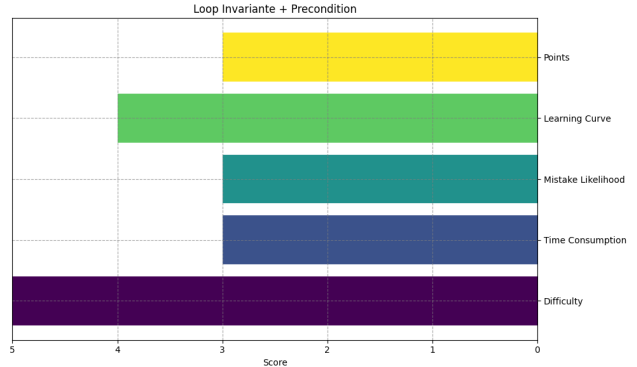
Raphaël Larisch @rlarisch · 10 months ago



I see your point, but not writing $v == 2$ will sacrifice our postcondition. So we are kind of in a situation where both cases are false, if I'm not mistaken. Maybe the question isn't exact enough after all.

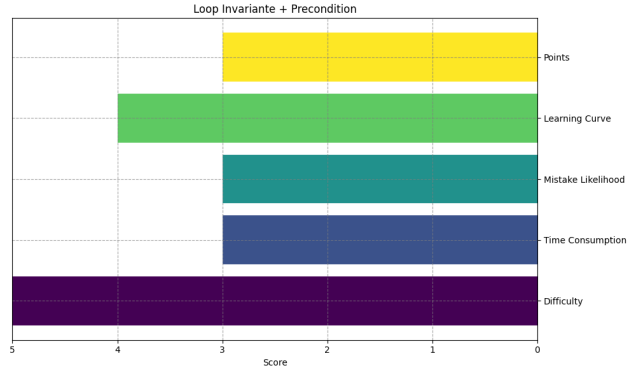
Loop Invariante + Precondition

- HS20 Prüfung + Implizit in FS22
- Gleiches Muster - Precondition so wählen, dass Postcondition folgt
- Trial & Error.



Loop Invariante + Precondition

- HS20 Prüfung + Implizit in FS22
- Gleiches Muster - Precondition so wählen, dass Postcondition folgt
- Trial & Error.



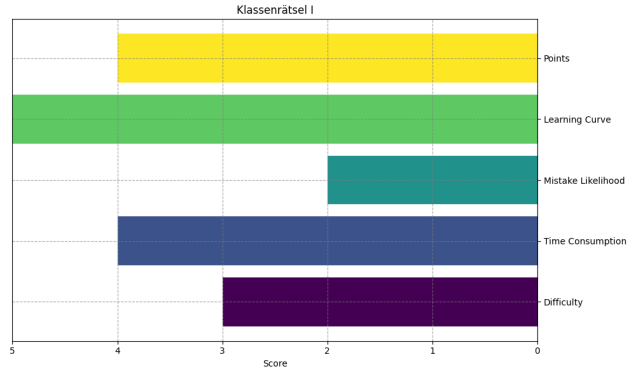
Klassenrätsel I

```
1  class A extends      {  
2      public int method1() {...}  
3      public int method2() {...}  
4  }  
5  class B extends      {  
6      public int method2() {...}  
7      public int method3() {...}  
8  }
```

```
1  A a = new A();  
2  A b = new B();  
3  
4  a.method1();  
5  int x = a.method2();  
6  int y = b.method2();  
7  
8  System.out.println(x);  
9  System.out.println(y);
```

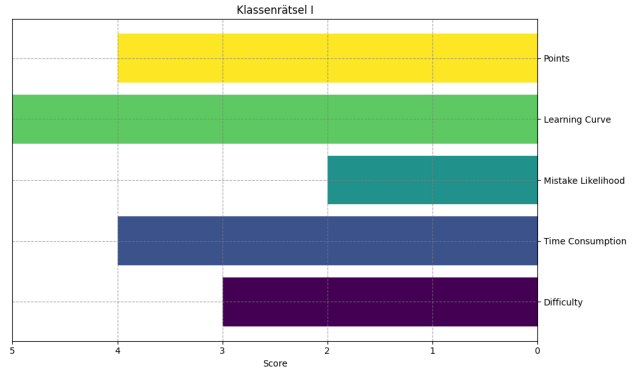
Klassenrätsel I

- Anfangs sehr schwer - Mit Übung eine der besten Aufgaben.
- Vererbungsbäume, Leuchtstifte helfen.
- Mühe mit Inheritance? [Summary](#) ↗.



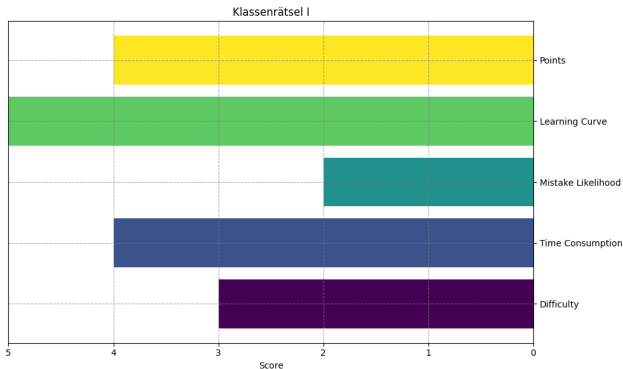
Klassenrätsel I

- Anfangs sehr schwer - Mit Übung eine der besten Aufgaben.
- Vererbungsbäume, Leuchtstifte helfen.
- Mühe mit Inheritance? [Summary](#) ↗.



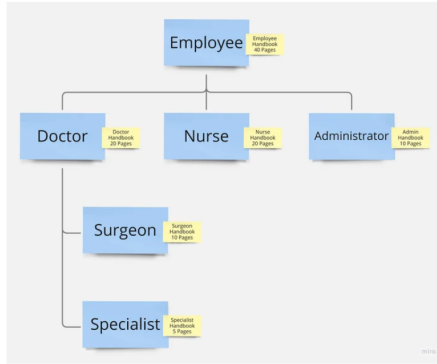
Klassenrätsel I

- Anfangs sehr schwer - Mit Übung eine der besten Aufgaben.
- Vererbungsbäume, Leuchtstifte helfen.
- Mühe mit Inheritance?
Summary [↗](#).



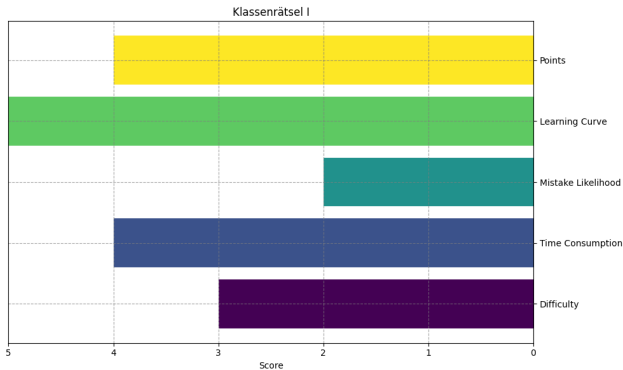
Inheritance

Real Life Example



Klassenrätsel I

- Anfangs sehr schwer - Mit Übung eine der besten Aufgaben.
- Vererbungsbäume, Leuchtstifte helfen.
- Mühe mit Inheritance? [Summary](#) ↗.



Klassenrätsel I

- Type beim Kompilieren ist der Referenztype.
- Type bei Runtime (bei Ausführung) ist der Type des Objekts.
- Attributwahl hängt ab vom Compile-Type.
- Methodenwahl hängt ab vom Runtime-Type.

$\underbrace{A}_1 \underbrace{a}_2 = \text{new } \underbrace{B}_3 ()$

1. Type der Referenzvariable - Referenztype
2. Name der Referenzvariable
3. Type des Objekts

Klassenrätsel I

- Type beim Kompilieren ist der Referenztype.
- Type bei Runtime (bei Ausführung) ist der Type des Objekts.
- Attributwahl hängt ab vom Compile-Type.
- Methodenwahl hängt ab vom Runtime-Type.

$\underbrace{A}_1 \underbrace{a}_2 = \text{new } \underbrace{B}_3 ()$

1. Type der Referenzvariable - Referenztype
2. Name der Referenzvariable
3. Type des Objekts

Klassenrätsel I

- Type beim Kompilieren ist der Referenztype.
- Type bei Runtime (bei Ausführung) ist der Type des Objekts.
- Attributwahl hängt ab vom Compile-Type.
- Methodenwahl hängt ab vom Runtime-Type.

$\underbrace{A}_1 \underbrace{a}_2 = \text{new } \underbrace{B}_3 ()$

1. Type der Referenzvariable - Referenztype
2. Name der Referenzvariable
3. Type des Objekts

Klassenrätsel I

- Type beim Kompilieren ist der Referenztype.
- Type bei Runtime (bei Ausführung) ist der Type des Objekts.
- Attributwahl hängt ab vom Compile-Type.
- Methodenwahl hängt ab vom Runtime-Type.

$\underbrace{A}_1 \underbrace{a}_2 = \text{new } \underbrace{B}_3 ()$

1. Type der Referenzvariable - Referenztype
2. Name der Referenzvariable
3. Type des Objekts

Klassenrätsel I

- Type beim Kompilieren ist der Referenztype.
- Type bei Runtime (bei Ausführung) ist der Type des Objekts.
- Attributwahl hängt ab vom Compile-Type.
- Methodenwahl¹ hängt ab vom Runtime-Type.

$\underbrace{A}_1 \underbrace{a}_2 = \text{new } \underbrace{B}_3 ()$

1. Type der Referenzvariable - Referenztype
2. Name der Referenzvariable
3. Type des Objekts

¹ausser static, private, final Methoden

Klassenrätsel II

```
1 Z c = new C();  
2 ((B)c).methodC();
```

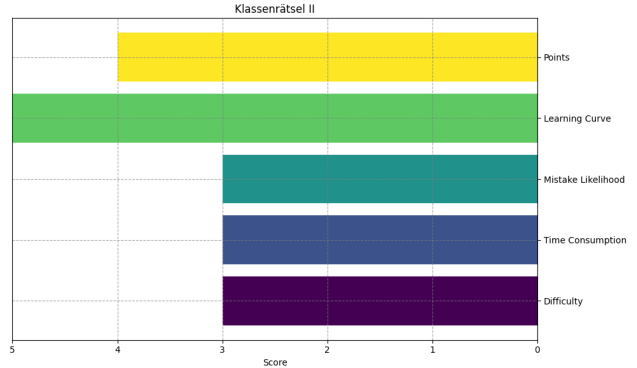
```
1 Z a = new A();  
2 a.methodA();
```

```
1 Animal miau = new Cat();  
2 miau.makeNoise();
```

```
1 Cat miep = new Animal();  
2 miep.makeNoise();
```

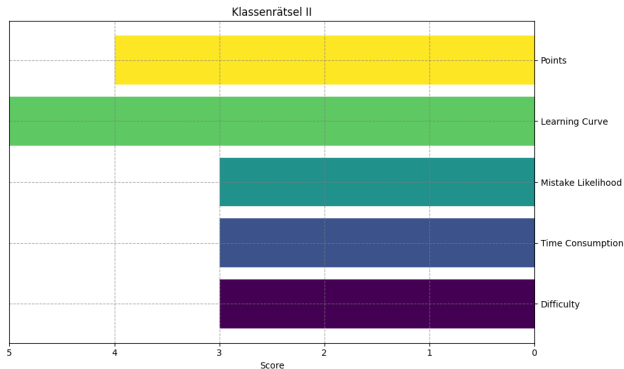
Klassenrätsel II

- Attributwahl mit Compile-Type.
- Methodenwahl mit Runtime-Type.
- Jede Methode und jedes Attribut wird einzeln betrachtet.
- Vererbung verstehen braucht Übung!



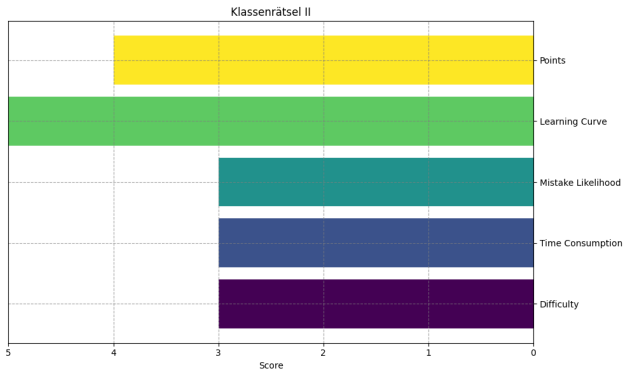
Klassenrätsel II

- Attributwahl mit Compile-Type.
- Methodenwahl mit Runtime-Type.
- Jede Methode und jedes Attribut wird einzeln betrachtet.
- Vererbung verstehen braucht Übung!



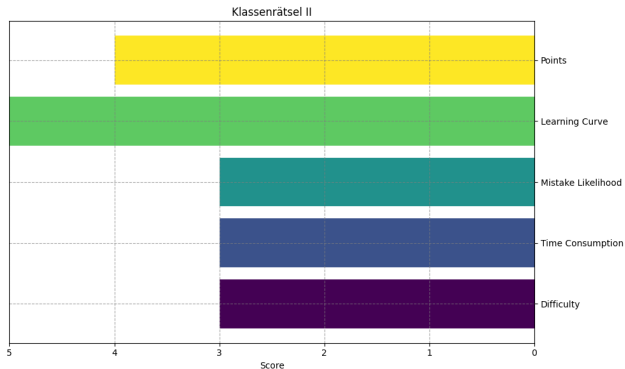
Klassenrätsel II

- Attributwahl mit Compile-Type.
- Methodenwahl mit Runtime-Type.
- Jede Methode und jedes Attribut wird einzeln betrachtet.
- Vererbung verstehen braucht Übung!



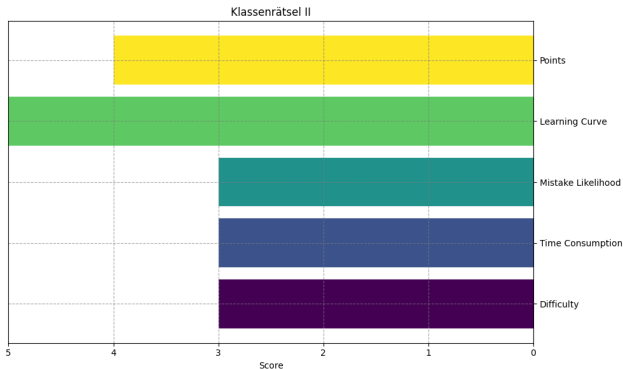
Klassenrätsel II

- Attributwahl mit Compile-Type.
- Methodenwahl mit Runtime-Type.
- Jede Methode und jedes Attribut wird einzeln betrachtet.
- Vererbung verstehen braucht Übung!



Klassenrätsel II

- Attributwahl mit Compile-Type.
- Methodenwahl mit Runtime-Type.
- Jede Methode und jedes Attribut wird einzeln betrachtet.
- Vererbung verstehen braucht Übung!



Klassenrätsel II

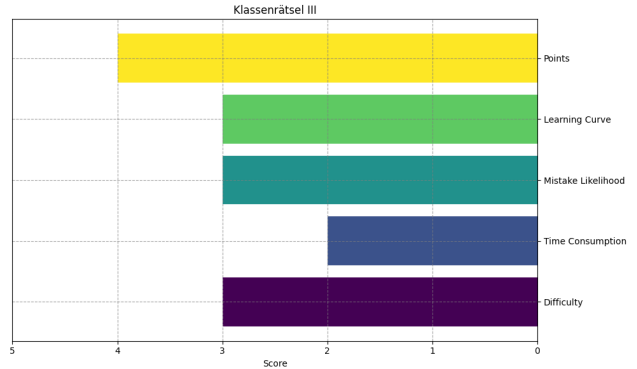
- Compiler-Fehler: Compile-Types unabhängig von Runtime-Types nicht kompatibel.
- Runtime-Exception (`ClassCastException`): Kein Compiler-Fehler und Runtime-Types sind nicht kompatibel.

Klassenrätsel III

```
1  public class OrganismTest {
2      public static void main(String[] args) {
3          Organism[] oArr = new Organism[] {
4              new Animal(),
5              new Flower(),
6              new Cat(),
7              new Cactus()
8          }
9          for(Organism o : oArr) {
10             o.printNameSpecialised();
11         }
12     }
13 }
```

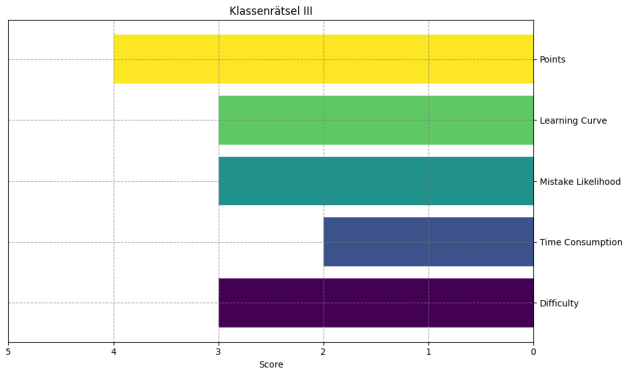
Klassenrätsel III

- Jedes Objekt einzeln betrachten.
- Einzelnes Objekt analog zu Klassenrätsel II lösen.



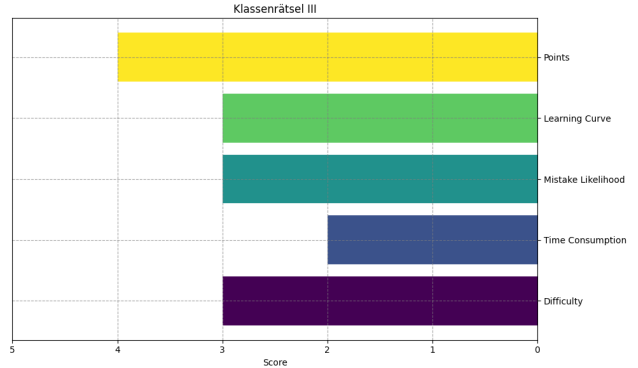
Klassenrätsel III

- Jedes Objekt einzeln betrachten.
- Einzelnes Objekt analog zu Klassenrätsel II lösen.



Klassenrätsel III

- Jedes Objekt einzeln betrachten.
- Einzelnes Objekt analog zu Klassenrätsel II lösen.



Klassenrätsel III

```
1  public class OrganismTest {
2      public static void main(String[] args) {
3          Organism[] oArr = new Organism[] {
4              new Animal(),
5              new Flower(),
6              new Cat(),
7              new Cactus()
8          }
9          for(Organism o : oArr) {
10             o.printNameSpecialised();
11         }
12     }
13 }
```

Klassenrätsel III

```
1  Organism a = new Animal();  
2  a.printNameSpecialised();
```

```
1  Organism c = new Cat();  
2  c.printNameSpecialised();
```

```
1  Organism f = new Flower();  
2  f.printNameSpecialised();
```

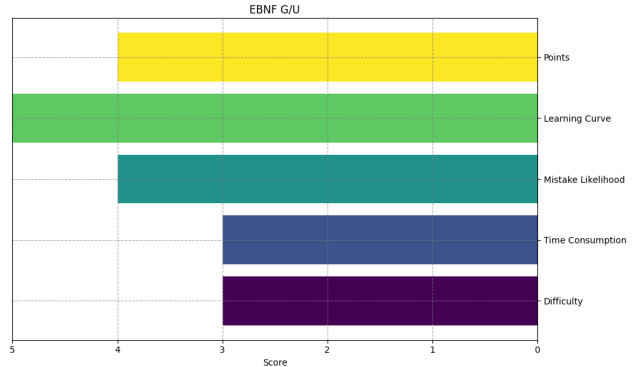
```
1  Organism k = new Cactus();  
2  k.printNameSpecialised();
```

EBNF G/U

<i>unary_expression</i>	\Leftarrow <i>unary_operator primary_expression</i>
<i>postfix_expression</i>	\Leftarrow <i>primary_expression</i> <i>postfix_expression</i> + + <i>postfix_expression</i> - -
<i>primary_expression</i>	\Leftarrow (<i>special_expression</i>) <i>identifier</i>
<i>unary_operator</i>	\Leftarrow * !
<i>assignment_expression</i>	\Leftarrow <i>expression assignment_operator special_expression</i>
<i>assignment_operator</i>	\Leftarrow = + =
<i>special_expression</i>	\Leftarrow <i>primary_expression</i> <i>special_expression binary_operator expression</i>
<i>binary_operator</i>	\Leftarrow * / + %
<i>expression</i>	\Leftarrow <i>assignment_expression</i> <i>postfix_expression</i> <i>unary_expression</i>

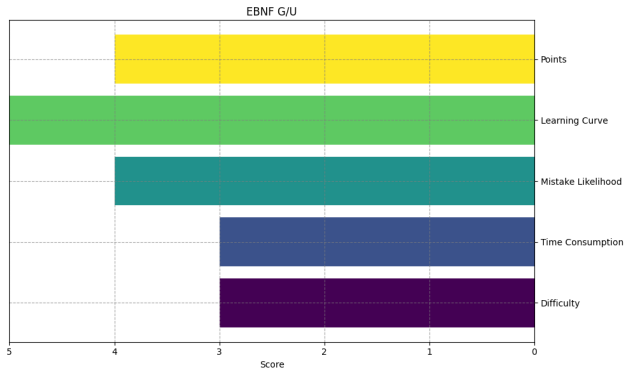
EBNF G/U

- EBNF Regeln kennen ist wichtig.
Summary Teil 1 [↗](#) Teil 2 [↗](#)
- Hier ist viel Übung nötig.
- Ausschlussverfahren meist schneller als Regel komplett prüfen.



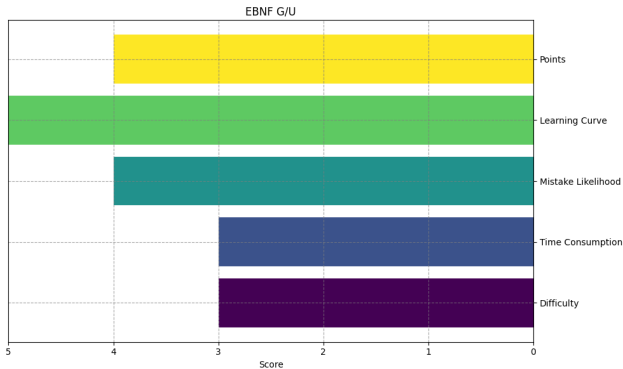
EBNF G/U

- EBNF Regeln kennen ist wichtig.
Summary Teil 1 [↗](#) Teil 2 [↗](#)
- Hier ist viel Übung nötig.
- Ausschlussverfahren meist schneller als Regel komplett prüfen.



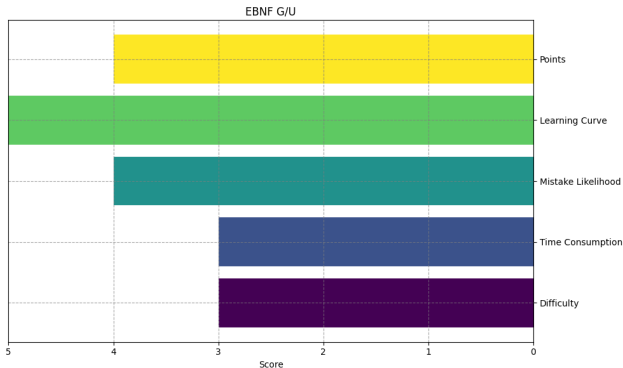
EBNF G/U

- EBNF Regeln kennen ist wichtig.
Summary Teil 1 [↗](#) Teil 2 [↗](#)
- Hier ist viel Übung nötig.
- Ausschlussverfahren meist
schneller als Regel komplett
prüfen.



EBNF G/U

- EBNF Regeln kennen ist wichtig.
Summary Teil 1 [↗](#) Teil 2 [↗](#)
- Hier ist viel Übung nötig.
- Ausschliessverfahren meist schneller als Regel komplett prüfen.

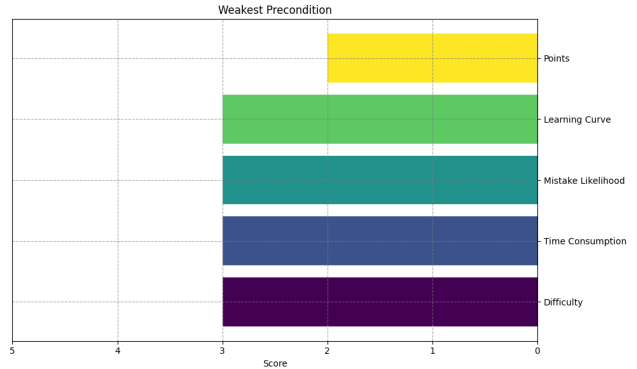


Weakest Precondition

```
1  WP : {                                     }  
2  x = 1 + y;  
3  z = 2 * x - 5 * z;  
4  a = x + y + z;  
5  Q : {a < 100 && a > 20}
```

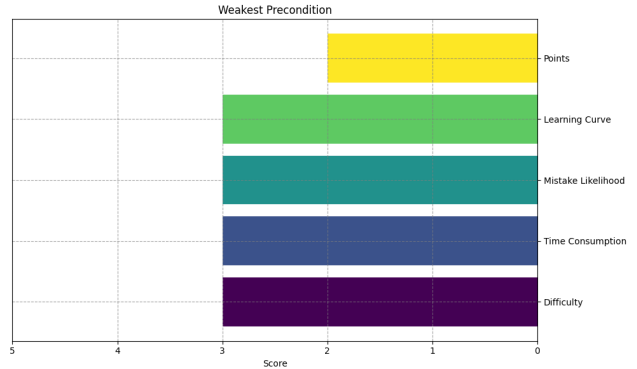
Weakest Precondition

- Rückwärtseinsetzen hilft hier
Summary [↗](#)
- Macht immer einen Reality-Check
eurer Lösung.



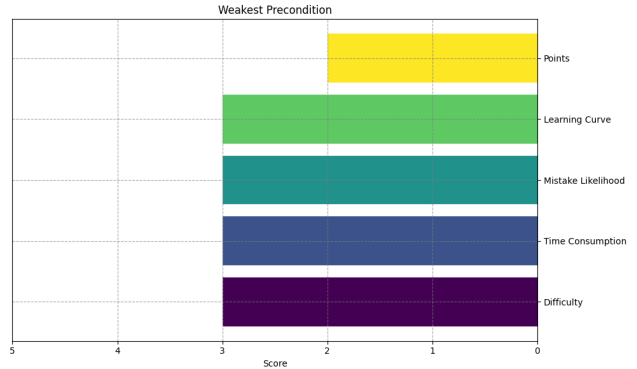
Weakest Precondition

- Rückwärtseinsetzen hilft hier
Summary [↗](#)
- Macht immer einen Reality-Check
eurer Lösung.



Weakest Precondition

- Rückwärtseinsetzen hilft hier
Summary [↗](#)
- Macht immer einen Reality-Check
eurer Lösung.

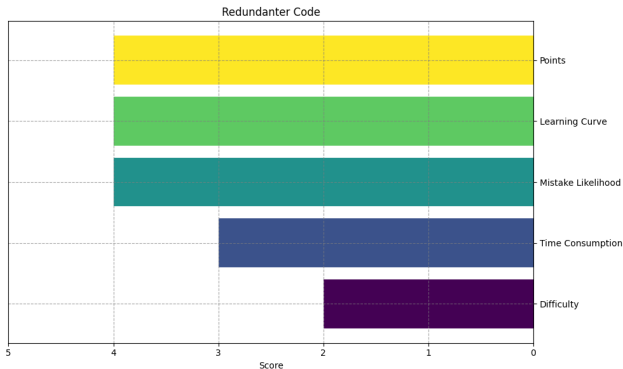


Redundanter Code

```
1  int x = 0;                                \\A
2  int y = x + (new Random()).nextInt(4);    \\B
3
4  if (y == 2) {                              \\C
5      int z = 23;                            \\D
6  } else {                                  \\E
7      int z = 24;                            \\F
8  }
9
10 if (y == 7) {                              \\G
11     System.out.println("seven");           \\H
12 } else if (y == 3) {                       \\I
13     System.out.println("seven");           \\J
14 }
```

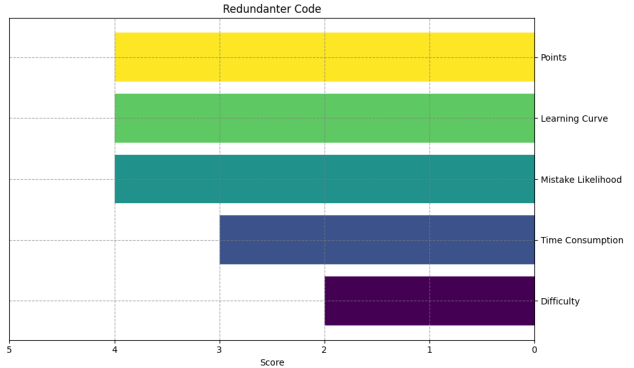

Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.



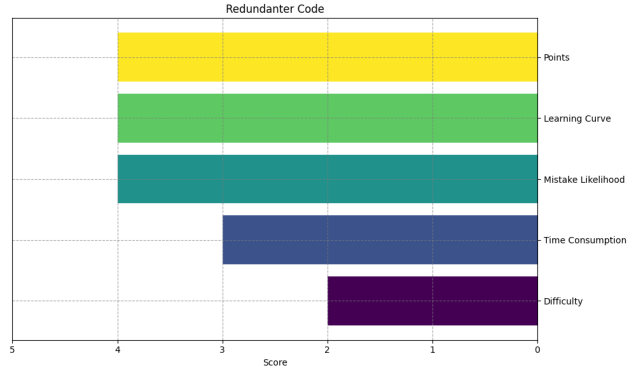
Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.



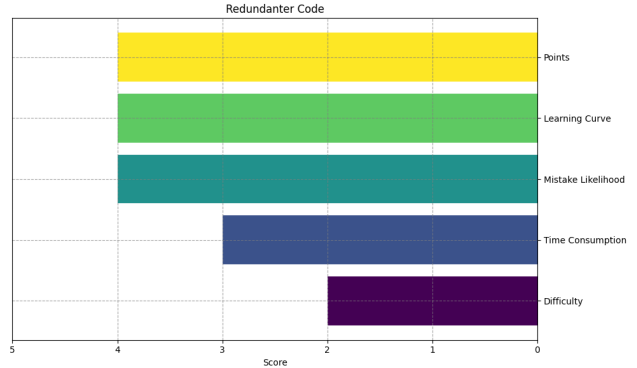
Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.



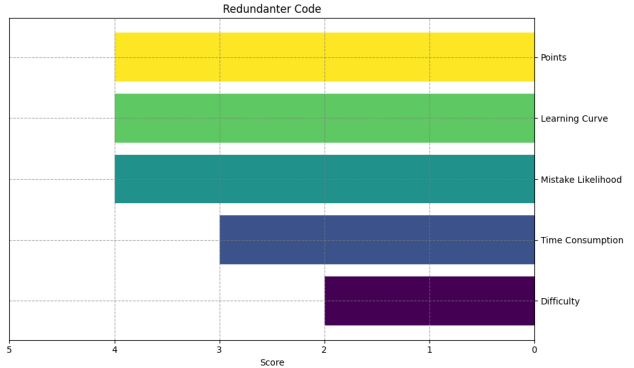
Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.



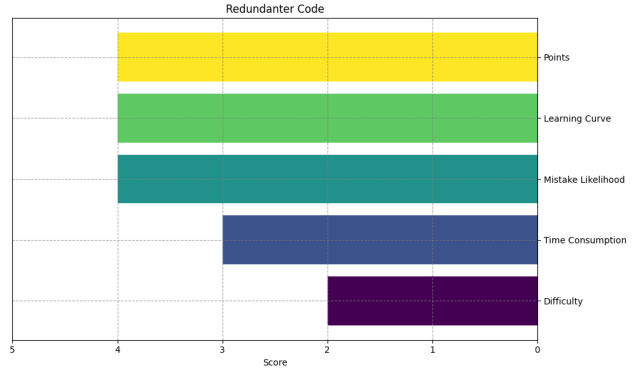
Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.



Redundanter Code

- Unbenutzte Variablen
- Variablen welche verändert aber nie benutzt werden im Resultat
- Addieren von Variablen mit Wert 0
- return statements welche nie erreicht werden
- Nicht zu schnell Schlüsse ziehen.

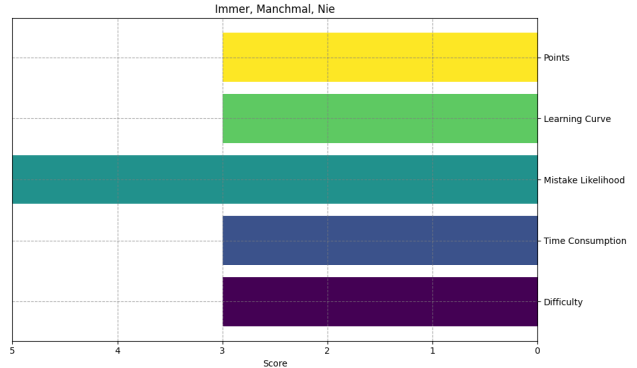


Immer, Manchmal, Nie

```
1  int x = 2;
2  int y = 0;
3  //A
4  while(x > y) {
5      //B
6      if(x = y + 1) {
7          //C
8          x = y;
9      } else {
10         //D
11         x = x + 10;
12         y = y + x / 2
13     }
14 }
```

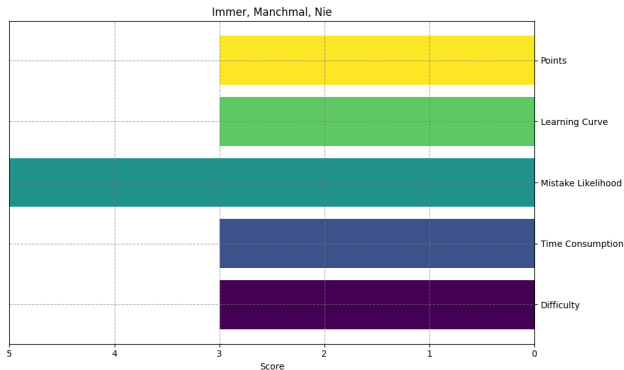
Immer, Manchmal, Nie

- Immer und Nie sind schwieriger als Manchmal.
- Immer / Nie meist wegen loop-condition, if/ else condition.
- Endliche Menge an Fallen - Übung ist sehr wertvoll hier.



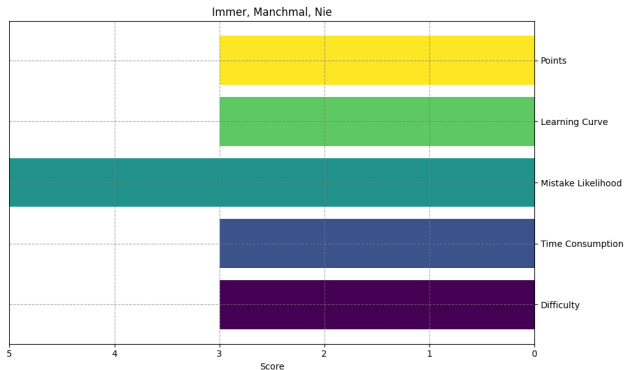
Immer, Manchmal, Nie

- Immer und Nie sind schwieriger als Manchmal.
- Immer / Nie meist wegen loop-condition, if/ else condition.
- Endliche Menge an Fallen - Übung ist sehr wertvoll hier.



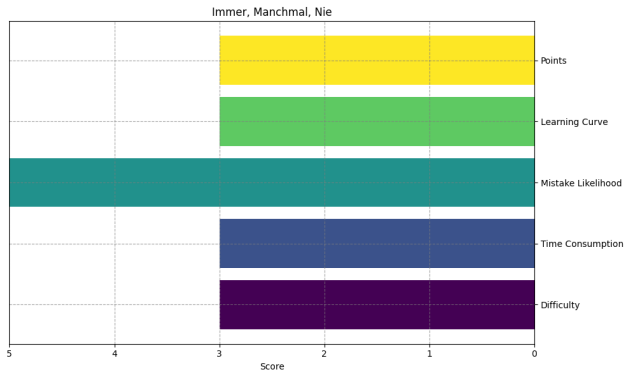
Immer, Manchmal, Nie

- Immer und Nie sind schwieriger als Manchmal.
- Immer / Nie meist wegen loop-condition, if/ else condition.
- Endliche Menge an Fallen - Übung ist sehr wertvoll hier.



Immer, Manchmal, Nie

- Immer und Nie sind schwieriger als Manchmal.
- Immer / Nie meist wegen loop-condition, if/ else condition.
- Endliche Menge an Fallen - Übung ist sehr wertvoll hier.



EBNF Äquivalente Regeln

Gegeben seien die EBNF-Beschreibungen *beispiel1* und *beispiel2*.

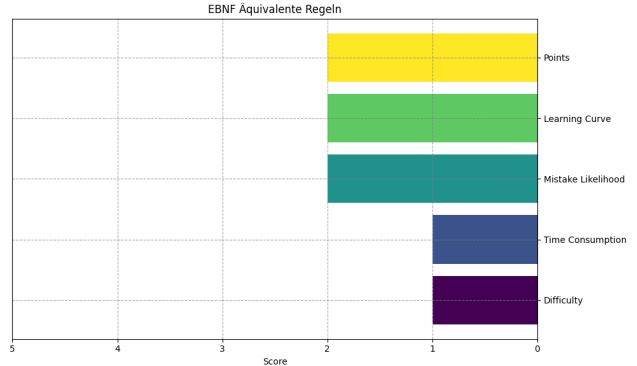
$$\textit{beispiel1} \Leftarrow \boxed{x} \textit{beispiel1} \mid \boxed{y}$$

$$\textit{beispiel2} \Leftarrow \boxed{x} \{ \boxed{x} \} \boxed{y}$$

Abbildung 1: **EBNF-Beschreibungen** von *beispiel1* und von *beispiel2*

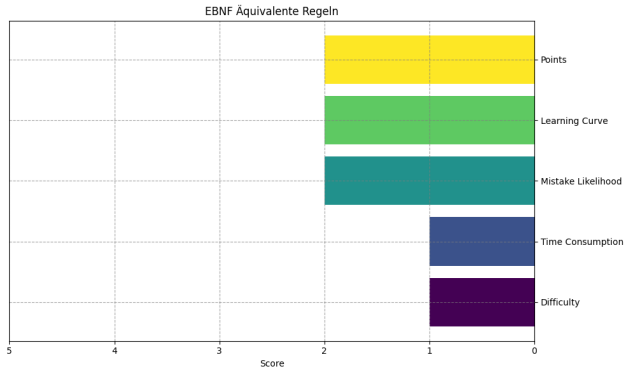
EBNF Äquivalente Regeln

- Meistens ein Gegenbeispiel
- Kandidaten für Gegenbeispiele einfach halten - leeres Wort, kurze Wörter
- Keine Beweise aus Diskreter Mathematik



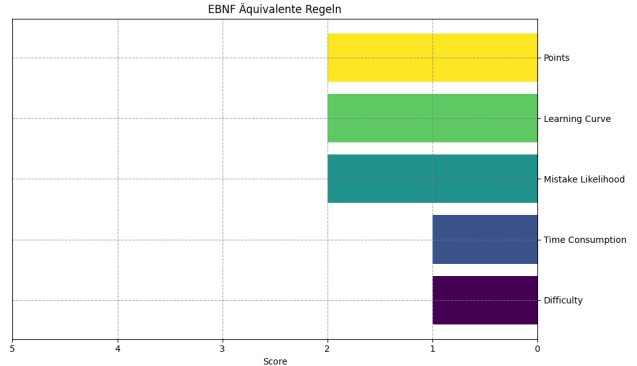
EBNF Äquivalente Regeln

- Meistens ein Gegenbeispiel
- Kandidaten für Gegenbeispiele einfach halten - leeres Wort, kurze Wörter
- Keine Beweise aus Diskreter Mathematik



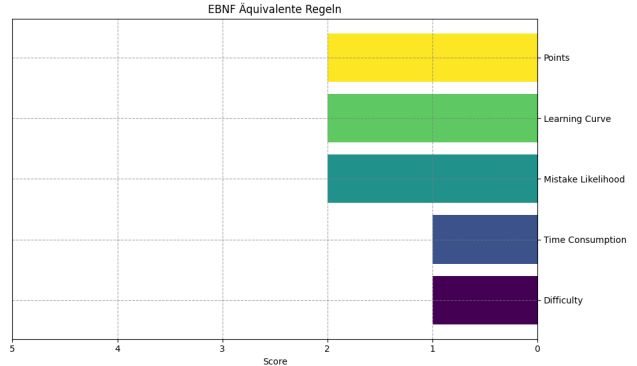
EBNF Äquivalente Regeln

- Meistens ein Gegenbeispiel
- Kandidaten für Gegenbeispiele einfach halten - leeres Wort, kurze Wörter
- Keine Beweise aus Diskreter Mathematik



EBNF Äquivalente Regeln

- Meistens ein Gegenbeispiel
- Kandidaten für Gegenbeispiele einfach halten - leeres Wort, kurze Wörter
- Keine Beweise aus Diskreter Mathematik



Hoare Triple

```
1 { x > 0 }  
2   y = x * 2;  
3 { y > 0 }
```

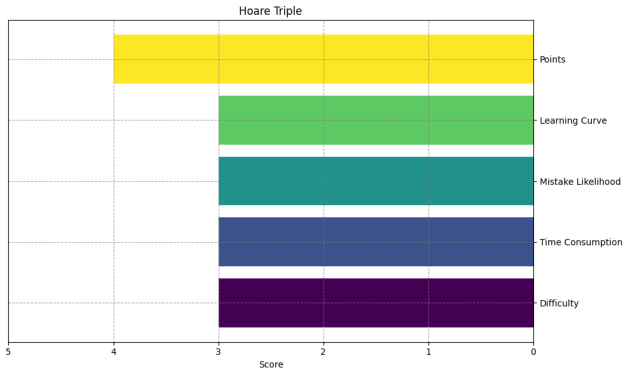
```
1 { true }  
2   z = x * x;  
3 { z > x }
```

```
1 { (a > 2) && (b == 3) }  
2   z = a + b;  
3 { z > 0 }
```

```
1 { a > 0 }  
2   y = a * a; z = y % a;  
3 { z > 1 }
```

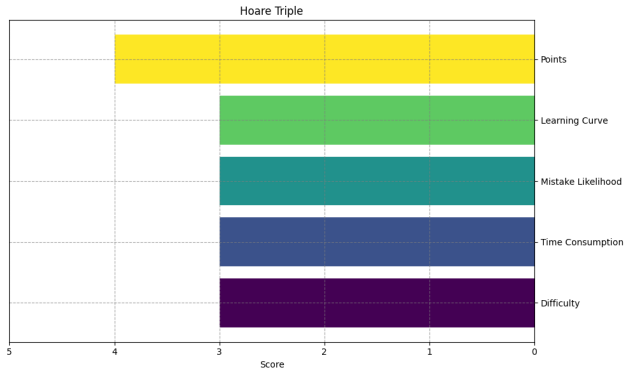
Hoare Triple

- Rückwärtseinsetzen und Vorwärtseinsetzen [↗](#)
- Gegenbeispiele sind oft die Grenzen selbst - bei $x > 0$ ist 0 die Grenze.
- 0, 1, -1 sind ebenfalls oft Gegenbeispiele
- Korrektheit vom Hoare Triple schwieriger als Gegenbeispiel finden



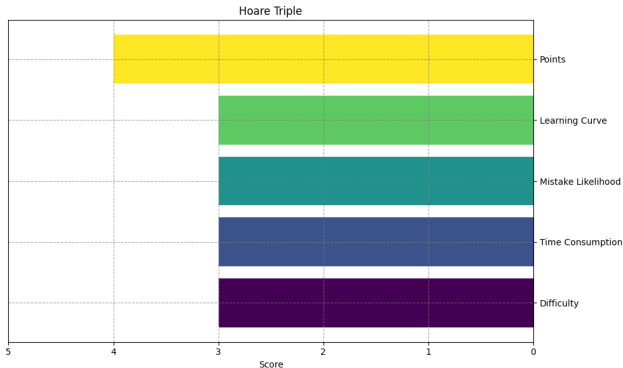
Hoare Triple

- Rückwärtseinsetzen und Vorwärtseinsetzen [!\[\]\(6302aad5aed157b291fddf37b4870784_img.jpg\)](#)
- Gegenbeispiele sind oft die Grenzen selbst - bei $x > 0$ ist 0 die Grenze.
- 0, 1, -1 sind ebenfalls oft Gegenbeispiele
- Korrektheit vom Hoare Triple schwieriger als Gegenbeispiel finden



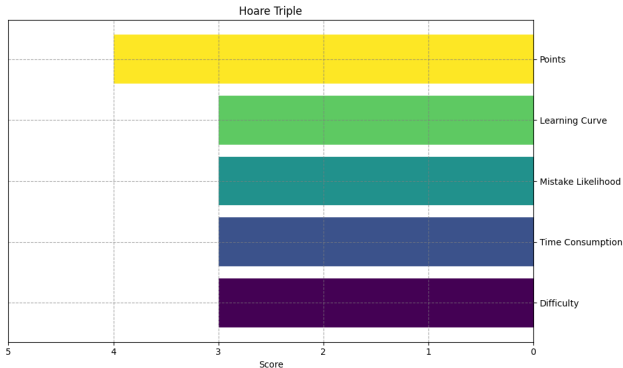
Hoare Triple

- Rückwärtseinsetzen und Vorwärtseinsetzen [↗](#)
- Gegenbeispiele sind oft die Grenzen selbst - bei $x > 0$ ist 0 die Grenze.
- 0, 1, -1 sind ebenfalls oft Gegenbeispiele
- Korrektheit vom Hoare Triple schwieriger als Gegenbeispiel finden



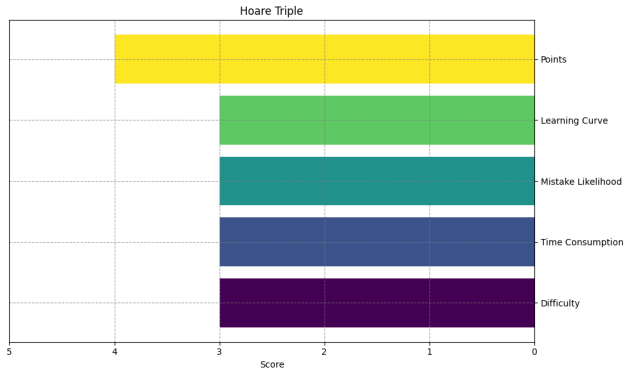
Hoare Triple

- Rückwärtseinsetzen und Vorwärtseinsetzen [!\[\]\(3da2b303d29c1ea489bbe26a3f5ac664_img.jpg\)](#)
- Gegenbeispiele sind oft die Grenzen selbst - bei $x > 0$ ist 0 die Grenze.
- 0, 1, -1 sind ebenfalls oft Gegenbeispiele
- Korrektheit vom Hoare Triple schwieriger als Gegenbeispiel finden



Hoare Triple

- Rückwärtseinsetzen und Vorwärtseinsetzen [!\[\]\(849840539e55921a3851a4ff96d7400d_img.jpg\)](#)
- Gegenbeispiele sind oft die Grenzen selbst - bei $x > 0$ ist 0 die Grenze.
- 0, 1, -1 sind ebenfalls oft Gegenbeispiele
- Korrektheit vom Hoare Triple schwieriger als Gegenbeispiel finden



Das kommt nie an der Prüfung - eben doch!

Gegeben sei die Klasse Cache:

```
import java.util.HashMap;

class Cache {
    final int CAPACITY = 2;
    HashMap<Integer, Integer> data =
        new LinkedHashMap<Integer, Integer> ();

    void add(int value) {
        if (data.size() >= CAPACITY) {
            int victim_count = Integer.MAX_VALUE;
            Integer victim = null;
            Set<Integer> keys = data.keySet();
            Iterator it = keys.iterator();

            while (it.hasNext()) {
                Integer next = (Integer) it.next();
                if (data.get(next) < victim_count) {
                    victim = next;
                    victim_count = data.get(next);
                }
            }
            data.remove(victim);
            data.put(value, 1);
        } else {
            data.put(value, 1);
        }
    } //add

    void access(int value) {
        Integer now = data.get(value);
        if (now == null) {
            add(value);
        } else {
            data.put(value, now+1);
        }
    } // access

    void print() {
        data.entrySet().forEach(entry -> {
            System.out.println("Key : " + entry.getKey() +
                               " Value : " + entry.getValue());
        });
    } // print
} // Cache
```

Exemplare von Cache enthalten die letzten beiden Objekte, auf die mittels `access(int value)` zugegriffen wurde (der Einfachheit halber verwenden wir hier nur Integer Objekte, wobei hier sichergestellt ist, dass Boxing und Unboxing keine Probleme liefern).

Das kommt nie an der Prüfung - eben doch!

- Generell ist der ganze Stoff Prüfungsrelevant.
- Es ist nützlich mehr als nur das mindeste, zu können.
- Informiert euch wie Datentypen funktionieren, wenn ihr sie braucht.
- HashMaps, HashSet, TreeSet, TreeMap, ArrayList, LinkedList, Queue, Stack, usw.

Das kommt nie an der Prüfung - eben doch!

- Generell ist der ganze Stoff Prüfungsrelevant.
- Es ist nützlich mehr als nur das mindeste, zu können.
- Informiert euch wie Datentypen funktionieren, wenn ihr sie braucht.
- HashMaps, HashSet, TreeSet, TreeMap, ArrayList, LinkedList, Queue, Stack, usw.

Das kommt nie an der Prüfung - eben doch!

- Generell ist der ganze Stoff Prüfungsrelevant.
- Es ist nützlich mehr als nur das mindeste, zu können.
- Informiert euch wie Datentypen funktionieren, wenn ihr sie braucht.
- HashMaps, HashSet, TreeSet, TreeMap, ArrayList, LinkedList, Queue, Stack, usw.

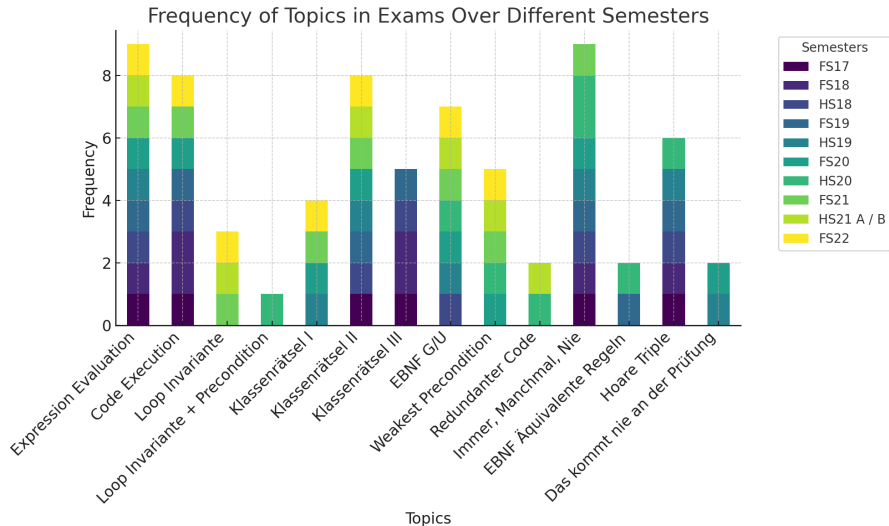
Das kommt nie an der Prüfung - eben doch!

- Generell ist der ganze Stoff Prüfungsrelevant.
- Es ist nützlich mehr als nur das mindeste, zu können.
- Informiert euch wie Datentypen funktionieren, wenn ihr sie braucht.
- HashMaps, HashSet, TreeSet, TreeMap, ArrayList, LinkedList, Queue, Stack, usw.







Das kommt nie an der Prüfung - eben doch!

- Generell ist der ganze Stoff Prüfungsrelevant.
- Es ist nützlich mehr als nur das mindeste, zu können.
- Informiert euch wie Datentypen funktionieren, wenn ihr sie braucht.
- HashMaps, HashSet, TreeSet, TreeMap, ArrayList, LinkedList, Queue, Stack, usw.


Statistics




Zusätzliche Übungen

- EBNF Übungen: Gohar  eprog.ch  eprog.ch 
- Weakest Precondition Übungen: Gohar 
- Hoare Triple Übungen: Gohar 
- Loop Invarianten Übungen: Gohar 

Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.


Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.


Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.


Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.


Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.

Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.

Final Words

- VIS Exams Collection 
- Übung macht den Meister!
- Fängt früh an - lieber 30 Minuten jeden Tag als alles in einer Woche.
- Macht eine Liste von typischen Fehlern.
- Löst die Aufgaben auch auf Zeit.
- Lässt eine Prüfung stehen - Für ein paar Tage vor der Prüfung.