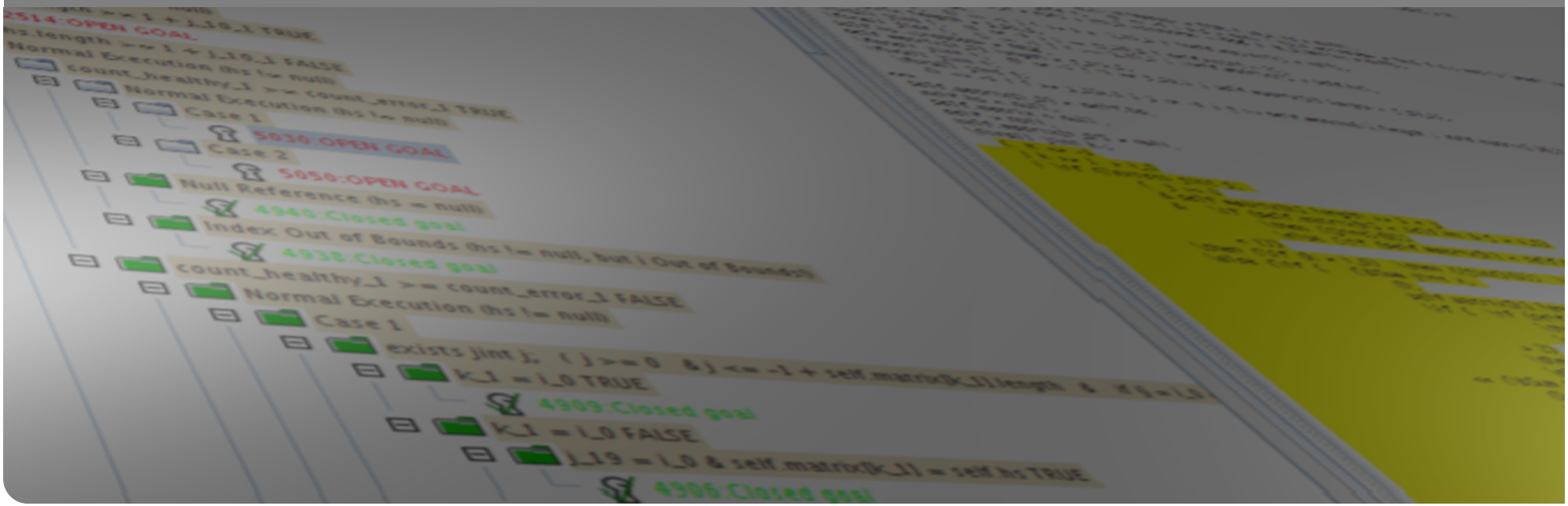
# Formale Systeme

## Prof. Dr. Bernhard Beckert, WS 2017/2018

Java Modeling Language (JML)

KIT – Institut für Theoretische Informatik



# Java Modeling Language

## Historie

# Java Modeling Language

## Historie

- Initiator Gary Leavens

# Java Modeling Language

## Historie

- Initiator Gary Leavens
- Erste Publikation 1999

# Java Modeling Language

## Historie

- Initiator Gary Leavens
- Erste Publikation 1999
- Seither kontinuierlicher Aufbau einer weltweiten Community

# Java Modeling Language

## Historie

- Initiator Gary Leavens
- Erste Publikation 1999
- Seither kontinuierlicher Aufbau einer weltweiten Community
- Standardisierungsvorhaben seit 2016

# Java Modeling Language

## Historie

- Initiator Gary Leavens
- Erste Publikation 1999
- Seither kontinuierlicher Aufbau einer weltweiten Community
- Standardisierungsvorhaben seit 2016

## Grundidee

Design by contract

# Java Modeling Language

## Historie

- Initiator Gary Leavens
- Erste Publikation 1999
- Seither kontinuierlicher Aufbau einer weltweiten Community
- Standardisierungsvorhaben seit 2016

## Grundidee

Design by contract

## JML Einführung, weitere Informationen

`http://www.key-project.org/thebook2/`, Chapter 7

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                  rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                    rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

JML-Annotationen sind spezielle Kommentare im Quelltext

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                   rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

## Vorbedingung

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                   rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

## Nachbedingung

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                   rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

Normale Terminierung: Terminierung und keine Ausnahme wird ausgelöst (*no exception thrown*)

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                    rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

## Invariante

```
public class PostInc {
 PostInc rec;
 int x, y;
 /*@  public invariant x     >= 0 && y     >= 0 &&
   @                   rec.x >= 0 && rec.y >= 0;
   @*/

 /*@ public normal_behavior
   @ requires true;
   @ ensures rec.x == \old(rec.y) &&
   @         rec.y == \old(rec.y) + 1;
   @*/
 public void postinc() {rec.x = rec.y++;}
}
```

## Zugriff im Nachzustand auf den Wert eines JML-Ausdruck im Vorzustand

# JML-Ausdrücke

Das syntaktische Material, aus dem JML-Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

# JML-Ausdrücke

Das syntaktische Material, aus dem JML-Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

In der Klasse `PostInc` deklarierte Felder

# JML-Ausdrücke

Das syntaktische Material, aus dem JML-Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

Operationen und Literal aus den Java Datentypen `int` und `boolean`

# JML-Ausdrücke

Das syntaktische Material, aus dem JML-Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

## Anwendungsoperator

# JML-Ausdrücke

Das syntaktische Material, aus dem JML-Ausdrücke aufgebaut sind, stammt zum größten Teil aus dem umgebenden Java Programm.

```
x >= 0 && y >= 0 && rec.x >= 0 && rec.y >= 0
```

Wie in Java steht `x >= 0` für `this.x >= 0`

# Alternative zum `old` Operator

```
PostInc oldrec;

/*@ ...
  @ requires oldrecy == rec.y;
  @ ensures rec.x == oldrecy && rec.y == oldrecy+1;
```

wobei `oldrecy` im nachfolgenden Code nicht auftritt.

# Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `this.rec == null` gilt?

# Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `this.rec == null` gilt?

Es wird eine *NullPointerException* ausgelöst.

# Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `this.rec == null` gilt?

Es wird eine *NullPointerException* ausgelöst.

Somit würde die Methode ihren Vertrag nicht erfüllen, denn es wird normale Terminierung verlangt.

# Zeiger auf Null (null pointer)

Was passiert, wenn die Methode in einem Zustand aufgerufen wird, in dem `this.rec == null` gilt?

Es wird eine *NullPointerException* ausgelöst.

Somit würde die Methode ihren Vertrag nicht erfüllen, denn es wird normale Terminierung verlangt.

Kein Problem.
JML nimmt als Voreinstellung, als *default*, an, dass alle vorkommenden Attribute und Parameter mit einem Objekttyp vom Nullobjekt verschieden sind.

# Überschreiben der Voreinstellung

```
public class PostInc{
 public  /*@ nullable @*/ PostInc rec;
 public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @  (rec != null ==> rec.x>=0 && rec.y>=0);
    @*/
  /*@ public normal_behavior
    @ requires rec != null;
    @ ensures rec.x == \old(rec.y) &&
    @         rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

# Überschreiben der Voreinstellung

```
public class PostInc{
 public  /*@ nullable @*/ PostInc rec;
 public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @   (rec != null ==> rec.x>=0 && rec.y>=0);
    @*/
  /*@ public normal_behavior
    @ requires rec != null;
    @ ensures rec.x == \old(rec.y) &&
    @         rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

Syntax zum Überschreiben der Voreinstellung

# Überschreiben der Voreinstellung

```
public class PostInc{
 public  /*@ nullable @*/ PostInc rec;
 public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @  (rec != null ==> rec.x>=0 && rec.y>=0);
    @*/
  /*@ public normal_behavior
    @ requires rec != null;
    @ ensures rec.x == \old(rec.y) &&
    @         rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

Verstärkte Vorbedingung jetzt erforderlich

# Überschreiben der Voreinstellung

```
public class PostInc{
 public  /*@ nullable @*/ PostInc rec;
 public int x,y;
  /*@  public invariant x>=0 && y>=0 &&
    @  (rec != null ==> rec.x>=0 && rec.y>=0);
    @*/
  /*@ public normal_behavior
    @ requires rec != null;
    @ ensures rec.x == \old(rec.y) &&
    @         rec.y == \old(rec.y)+1;
    @*/
public void postinc() {rec.x = rec.y++;}}
```

Änderung der Invarianten erforderlich

# Übungsaufgabe

Was ist die richtige Nachbedingung?

```
public class PostIncxx{
 public PostInc rec;
 public int x;
  /*@  public invariant x>=0 && rec.x>=0;
    @*/

  /*@ public normal_behavior
    @ requires true;
    @ ensures ???;
    @*/
 public void postinc() {rec.x = rec.x++;}}
```

# Übungsaufgabe

## Lösung

```
public class PostIncxx{
 public PostInc rec;
 public int x;
  /*@  public invariant x>=0 && rec.x>=0;
    @*/

  /*@ public normal_behavior
    @ requires true;
    @ ensures  rec.x == \old(rec.x);
    @*/
 public void postinc() {rec.x = rec.x++;}}
```

```
class SITA {
 int[] a1, a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @     r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
public int commonEntry(int l, int r) {...} }
```

```
class SITA {
 int[] a1, a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @     r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
public int commonEntry(int l, int r) {...} }
```

Zwei Deklarationen von Nachbedingungen werde wie ihre Konjunktion behandelt.

```
class SITA {
 int[] a1, a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @     r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
public int commonEntry(int l, int r) {...} }
```

JML-Schlüsselwort für den Rückgabewert einer Methode, falls vorhanden.

```
class SITA {
 int[] a1, a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @     r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
public int commonEntry(int l, int r) {...} }
```

Die Methode `commonEntry` soll einen Index `i` im Intervall `[l,r)` finden, für den die beiden Felder `a1`,`a2` denselben Wert haben oder `i` ist gleich der rechten Suchgrenze.

```
class SITA {
 int[] a1, a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @     r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @    a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
public int commonEntry(int l, int r) {...} }
```

Außerdem, sollen für alle Indizes kleiner als *i* die Felder `a1,a2` verschiedene Werte haben.

# Quantoren in JML

## Syntax

```
(\forall C x; B; R)        (\exists C x; B; R)
```

`B` *Bereichseinschränkung*
`R` *Rumpf*

# Quantoren in JML

## Syntax

```
(\forall C x; B; R)        (\exists C x; B; R)
```

`B` *Bereichseinschränkung*
`R` *Rumpf*

## Bedeutung in prädikatenlogischer Notation

$$\forall x^C(B \rightarrow R) \qquad \exists x^C(B \wedge R)$$

# Quantoren in JML

## Syntax

```
(\forall C x; B; R)        (\exists C x; B; R)
```

`B` *Bereichseinschränkung*
`R` *Rumpf*

## Bedeutung in prädikatenlogischer Notation

$$\forall x^C(B \rightarrow R) \qquad \exists x^C(B \wedge R)$$

## Beispiel

`(\forall C x; B; R)` und
`(\forall C x; true; (B ==> R))`
sind äquivalent.

# Vergleich der Notationen

| JML | Prädikatenlogik |
|---|---|
| `==` | $\doteq$ |
| `&&` | $\wedge$ |
| `\|\|` | $\vee$ |
| `!` | $\neg$ |
| `==>` | $\rightarrow$ |
| `<==>` | $\leftrightarrow$ |
| `(\forall C x;e1;e2)` | $\forall x(\neg x \doteq \mathit{null} \wedge [e1]) \rightarrow [e2])$ |
| `(\exists C x;e1;e2)` | $\exists x(\neg x \doteq \mathit{null} \wedge [e1] \wedge [e2])$ |

Dabei steht $[ei]$ für die prädikatenlogische Notation des JML-Ausdrucks `ei`.

# Fortsetzung

```
class SITA {
 int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @       r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @     a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
 public int commonEntry(int l, int r) {...}}
```

# Fortsetzung

```
class SITA {
 int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @       r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @     a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
 public int commonEntry(int l, int r) {...}}
```

Vorbedingung: Einschränkungen an die Parameter.

# Fortsetzung

```
class SITA {
 int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @       r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @     a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
 public int commonEntry(int l, int r) {...}}
```

Die *assignable* Klausel gibt an, welche Werte die nachfolgende Methode höchstens ändern darf.

# Fortsetzung

```
class SITA {
 int[] a1,a2;
 /*@ public normal_behaviour
   @  requires 0 <= l && l < r &&
   @       r <= a1.length && r <= a2.length;
   @  ensures (l <= \result && \result < r &&
   @     a1[\result] ==  a2[\result])
   @     || \result == r;
   @  ensures (\forall int j; l <= j && j < \result;
   @     a1[j] !=  a2[j]);
   @  assignable \nothing;
   @*/
 public int commonEntry(int l, int r) {...}}
```

Die Methode `commonEntry` soll nichts ändern.
Sie ist eine *reine Methode* (*pure method*)

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

Schlüsselwort für Schleifeninvariante (*loop invariant*)

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

## Schleifeninvariante

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

*assignable* Klausel auch für Schleifen

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

## Variante, sichert Terminierung

# Schleifenspezifikationen

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ loop_invariant
    @   l <= k && k <= r &&
    @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
    @ assignable \nothing;
    @ decreases a1.length - k;
    @*/
  while(k < r && a1[k] != a2[k]) k++;
  return k;
}
```

Details folgen jetzt.

# Anforderungen an Schleifeninvarianten

Die Angabe einer Schleifeninvarianten *SI* verlangt, dass

1. (Anfangsfall)
   *SI* vor Eintritt in die Schleife gilt,

# Anforderungen an Schleifeninvarianten

Die Angabe einer Schleifeninvarianten *SI* verlangt, dass

1. (Anfangsfall)
   *SI* vor Eintritt in die Schleife gilt,
2. (Iterationsschritt)
   für jeden Programmzustand $s_0$, in dem *SI* und die Schleifenbedingung gilt, im Zustand $s_1$, der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit $s_0$ erreicht wird, wieder *SI* gilt.

# Anforderungen an Schleifeninvarianten

Die Angabe einer Schleifeninvarianten *SI* verlangt, dass

1. (Anfangsfall)
   *SI* vor Eintritt in die Schleife gilt,
2. (Iterationsschritt)
   für jeden Programmzustand $s_0$, in dem *SI* und die Schleifenbedingung gilt, im Zustand $s_1$, der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit $s_0$ erreicht wird, wieder *SI* gilt.
3. (Anwendungsfall)
   nach Beendigung der Schleife reicht die Schleifeninvariante *SI* zusammen mit den Bedingungen für die Schleifenterminierung aus, um die Nachbedingung zu beweisen.

# Anforderungen an Schleifeninvarianten

Die Angabe einer Schleifeninvarianten *SI* verlangt, dass

1. (Anfangsfall)
   *SI* vor Eintritt in die Schleife gilt,
2. (Iterationsschritt)
   für jeden Programmzustand $s_0$, in dem *SI* und die Schleifenbedingung gilt, im Zustand $s_1$, der nach einmaligen Ausführen des Schleifenrumpfes beginnend mit $s_0$ erreicht wird, wieder *SI* gilt.
3. (Anwendungsfall)
   nach Beendigung der Schleife reicht die Schleifeninvariante *SI* zusammen mit den Bedingungen für die Schleifenterminierung aus, um die Nachbedingung zu beweisen.

Sind diese drei Forderungen in unserem Beispiel erfüllt?

# Nachprüfung: Anfangsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
```

# Nachprüfung: Anfangsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
```

Die Schleifeninvariante wird zu

```
 l <= l && l <= r &&
(\forall int i; l<=i && i<l; a1[i] != a2[i]);
```

# Nachprüfung: Anfangsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
```

Die Schleifeninvariante wird zu

```
 l <= l && l <= r &&
(\forall int i; l<=i && i<l; a1[i] != a2[i]);
```

Triviale Identität

# Nachprüfung: Anfangsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
```

Die Schleifeninvariante wird zu

```
 l <= l && l <= r &&
(\forall int i; l<=i && i<l; a1[i] != a2[i]);
```

Folgt aus der Vorbedingung

# Nachprüfung: Anfangsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
```

Die Schleifeninvariante wird zu

```
 l <= l && l <= r &&
(\forall int i; l<=i && i<l; a1[i] != a2[i]);
```

Leerer Bereich des Quantors

# Nachprüfung: Iterationsschritt

```
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]); */
while(k < r && a1[k] != a2[k]) k++;
```

# Nachprüfung: Iterationsschritt

```
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]); */
while(k < r && a1[k] != a2[k]) k++;
```

## Invariante vor dem Schleifendurchlauf

```
l<=k && k<=r &&(\forall int i;l<=i && i<k; a1[i]!=a2[i])
```

# Nachprüfung: Iterationsschritt

```
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]); */
while(k < r && a1[k] != a2[k]) k++;
```

## Invariante vor dem Schleifendurchlauf

```
l<=k && k<=r &&(\forall int i;l<=i && i<k; a1[i]!=a2[i])
```

## Schleifenbedingung

```
k < r && a1[k]!=a2[k]
```

# Nachprüfung: Iterationsschritt

```
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]); */
while(k < r && a1[k] != a2[k]) k++;
```

## Invariante vor dem Schleifendurchlauf

```
l<=k && k<=r &&(\forall int i;l<=i && i<k; a1[i]!=a2[i])
```

## Schleifenbedingung

```
k < r && a1[k]!=a2[k]
```

## Inv. nach dem Schleifendurchlauf: $k \rightsquigarrow k + 1$

```
l<=k+1 && k+1<=r &&
(\forall int i;l<=i && i<k+1; a1[i]!=a2[i])
```

# Nachprüfung: Anwendungsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

# Nachprüfung: Anwendungsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

## Fallunterscheidung

# Nachprüfung: Anwendungsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

## Fallunterscheidung

1. Die Schleife terminiert, weil `k<r` nicht mehr gilt

# Nachprüfung: Anwendungsfall

```
int k = l;
/*@ loop_invariant  l <= k && k <= r &&
  @   (\forall int i; l<=i && i<k; a1[i] != a2[i]);
  @*/
while(k < r && a1[k] != a2[k]) k++;
return k;
```

## Fallunterscheidung

1. Die Schleife terminiert, weil `k<r` nicht mehr gilt
2. Es gilt `k<r`, und die Schleife terminiert, weil `a1[k] == a2[k]` gilt.

# Nachprüfung: Anwendungsfall

## Schleifeninvariante

```
l <= k && k <= r &&
(\forall int i; l<=i && i<k; a1[i] != a2[i]);
```

## Negierte Schleifenbedingung (Fall 1)

```
 k >= r
```

## Nachbedingung (nach Schleife und `return k;`)

```
((l<=\result && \result<r && a1[\result]==a2[\result])
     || \result == r)
 &&
(\forall int j; l<=j && j<\result;a1[j] != a2[j])
```

# Nachprüfung: Anwendungsfall

## Schleifeninvariante

```
l <= k && k <= r &&
(\forall int i; l<=i && i<k; a1[i] != a2[i]);
```

## Negierte Schleifenbedingung (Fall 2)

```
 k < r && a1[k] == a2[k]
```

## Nachbedingung (nach Schleife und `return k;`)

```
((l<=\result && \result<r && a1[\result]==a2[\result])
     || \result == r)
 &&
(\forall int j; l<=j && j<\result;a1[j] != a2[j])
```

## Terminierung

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ decreases a1.length - k; */
  while(k < r && a1[k] != a2[k]) k++;
}
```

## Terminierung

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ decreases a1.length - k; */
  while(k < r && a1[k] != a2[k]) k++;
}
```

Der Ausdruck nach dem Schlüsselwort `decreases` heißt die *Variante* der Schleife.

## Terminierung

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ decreases a1.length - k; */
  while(k < r && a1[k] != a2[k]) k++;
}
```

Die Variante ist stets $\geq 0$

## Terminierung

```
public int commonEntry(int l, int r) {
  int k = l;
  /*@ decreases a1.length - k; */
  while(k < r && a1[k] != a2[k]) k++;
}
```

Die Variante wird in jedem Schleifendurchlauf echt kleiner.

# Verallgemeinerte Quantoren in JML

## Generalized Quantifiers

```
(\sum       T x; R ; t)
(\product   T x; R ; t)
(\max       T x; R ; t)
(\min       T x; R ; t)
```

# Verallgemeinerte Quantoren in JML

Generalized Quantifiers

```
(\sum       T x; R ; t)
(\product   T x; R ; t)
(\max       T x; R ; t)
(\min       T x; R ; t)
```

## Beispiele

```
(\sum     int i; 0<=i && i<5; i)    == 0+1+2+3+4
(\product int i; 0< i && i<5; i)    == 1*2*3*4
(\max     int i; 0<=i && i<5; i)    == 4
(\min     int i; 0<=i && i<5; i-1) == -1
```

# Methodenvertrag: Weiteres Beispiel

```
class SumAndMax {
  int sum; int max;
/*@ public normal_behaviour
  @  requires (\forall int i;0<=i && i<a.length;0<=a[i]);
  @  assignable sum, max;
  @  ensures (\forall int i;0<=i && i<a.length;a[i]<=max);
  @  ensures (a.length > 0
  @    ==> (\exists int i;0<=i && i<a.length;max == a[i]));
  @  ensures sum == (\sum int i;0<=i && i <a.length; a[i]);
  @  ensures sum <= a.length * max;
  @*/
 void sumAndMax(int[] a) { ....}}
```