Kapitel 07

Variablen und deren Gültigkeit

Fachgebiet Eingebettete Systeme und ihre Anwendungen
Prof. Dr. Andreas Koch

TECHNISCHE UNIVERSITÄT DARMSTADT

Inhalt des 7. Kapitels
Variablen und deren Gültigkeit

7.1 Konstanten und Variablen

- Objekt- und Klassenvariablen
- Was ist eine Konstante?
- Objekt- und Klassenkonstanten

7.2 Gültigkeit von Variablen

- Scope
- Lebenszeit
- Garbage Collection

7.1 Konstanten und Variablen
Objekt- und Klassenvariablen

- Bei den Datenkomponenten einer Klasse gibt es die Unterscheidung zwischen:
  - *Objekt- und Klassenvariable*
  - *Objekt- und Klassenkonstanten*

- *Syntaktische Unterscheidung*:
  - Klassenvariablen bzw. –konstanten werden durch das Schlüsselwort **`static`** deklariert.

Objekt- und Klassenvariablen

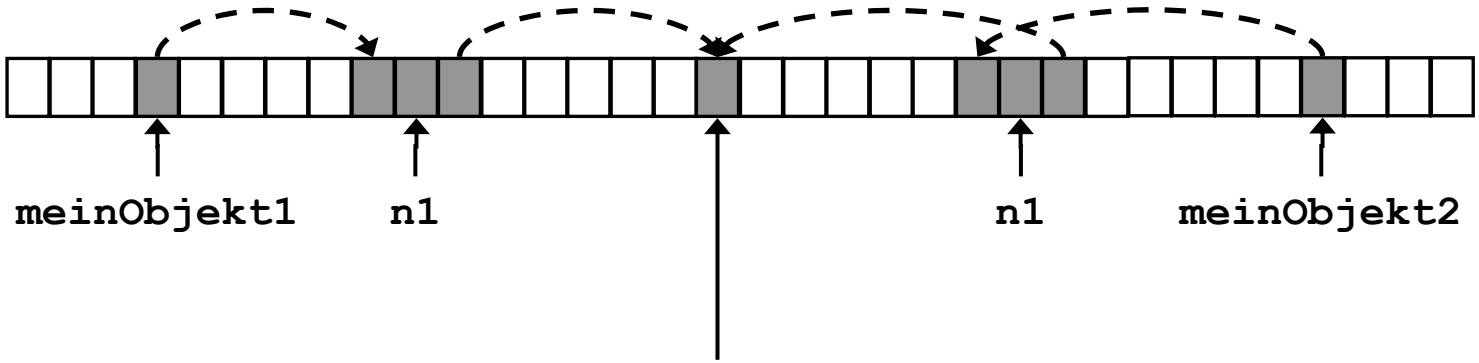
- *Semantische Unterscheidung*:
  - Eine Klassenvariable bzw. –konstante ist ein einzelnes und einmaliges Objekt. Sie werden in der **Klasse** selber und nicht im Objekt gespeichert.
  - Eine Objektvariable bzw. –konstante gibt es einmal pro erzeugtem **Objekt** (Instanz) der Klasse.
  - Die Bestandteile von Objekten sind also die Objektvariablen.
  - Auf Klassenvariablen bzw. –konstanten kann man auch ohne konkretes Objekt der Klasse zugreifen.
    - über „***`Klassenname.Klassenvariable`***“
    - Hinweis: Die gilt analog auch später für die Klassenmethoden.

Objekt- und Klassenvariablen Beispiel

```
public class MeineKlasse
   {
        public int n1;                                  // Objektvariable
        public static int n2;                           // Klassenvariable
   }

...
MeineKlasse meinObjekt1 = new MeineKlasse();
MeineKlasse meinObjekt2 = new MeineKlasse();

meinObjekt1.n1 = 1;
meinObjekt1.n2 = 2;
meinObjekt2.n1 = 3;
meinObjekt2.n2 = 4;

System.out.println ( meinObjekt2.n1 );          // -> 3
System.out.println ( meinObjekt2.n2 );          // -> 4
System.out.println ( meinObjekt1.n1 );          // -> 1
System.out.println ( meinObjekt1.n2 );          // -> 4(!)
System.out.println ( MeineKlasse.n2 );          // -> 4(!)
```

Objekt- und Klassenvariablen Beispiel

- *Speichersicht*



- *Erläuterungen*
  - **`meinObjekt1.n1`** und **`meinObjekt2.n1`** sind **zwei separate int–Variablen**, die Bestandteile der Objekte **`meinObjekt1`** bzw. **`meinObjekt2`** sind.

Objekt- und Klassenvariablen Beispiel

- *weitere Erläuterungen*
  - **`MeineKlasse.n2`** ist im Gegensatz dazu nur **ein einzelne, isolierte int–Variable,** die ein einziges Mal irgendwo im Speicher angelegt wird und für alle Objekte des Typs **`MeineKlasse`** gilt.

    → **`meinObjekt1.n2`** und **`meinObjekt2.n2`** bezeichnen denselben **`int`**–Wert.

    → daher ist eine Änderung der Klassenvariable **`n2`** für alle **`MeineKlasse`**-Objekte gültig
  - Die letzte Zeile auf der vorherigen Folie zeigt, wie man ohne ein Objekt von **`MeineKlasse`** auf dieses einzelne Objekt **`n2`** zugreifen kann.

Objekt- und Klassenvariablen Beispiel

- *Realisierung während der Kompilierung*
  - Beim Übersetzen haben die beiden Ausdrücke **`meinObjekt1.n1`** und **`meinObjekt1.n2`** für den Compiler unterschiedliche Bedeutungen.
  - In beiden Fällen konstruiert der Compiler Java Byte Code. Die Adresse der Objekte
    - **`meinObjekt1.n1`**
    - **`meinObjekt1.n2`**

    wird dabei unterschiedlich berechnet!
    - Bei **`meinObjekt1.n1`** wird die Adresse berechnet, indem die Position von **`n1`** in **`MeineKlasse`** auf den Wert von **`meinObjekt1`** aufaddiert wird.
    - Bei **`meinObjekt1.n2`** wird eine globale Adresse direkt eingesetzt. Der Compiler hat sich natürlich irgendwo intern die Adresse von MeineKlasse.n2 gemerkt.

Was ist eine Konstante?

- *Erinnerung*:
  Eine Variable ist ein symbolischer Name für eine Speicheradresse.

- *Beispiel*: Eine Zeichenvariable **`var`**, die ein Zeichen speichern soll, kann man einrichten und sofort mit dem Zeichen **`,a‘`** initialisieren.

```
char var = 'a';
```

Konstante Deklaration:

```
final char var = 'a';
```

Konstante

Konstanten sind Variablen deren Wert während der Laufzeit nicht mehr verändert werden kann.

Eine **Konstante** wird ähnlich wie eine Variable deklariert, mit folgenden zwei Unterschieden:

- Sie enthält das Schlüsselwort **`final`** vor dem Typnamen und
- *muss* unmittelbar bei der Deklaration initialisiert werden.

Sinn von Konstanten

Was ist der Sinn Variablen als konstant zu deklarieren?

- Oft ist ein Objekt von seiner inneren Logik her wirklich konstant.
- *Beispiele*:

```
final float pi = (float) 3.14159;
final char waehrung = '$';
```

- Mit **`final`** kann man verhindern, dass der Wert irgendwo im Quellcode aus Versehen überschrieben wird (Fehler beim Kompilieren).
- Man muss sich den Wert der Konstanten während der Programmierung nicht merken.
- Die Konsistenz wird sichergestellt.
- Konstanten müssen bei einer Anpassung des Programms aber nur einmal geändert werden (z.B. Änderung der Währung).

Konstanten und Literale

- *Erinnerung*: Zeichenketten der folgenden Formen
  - 69534
  - 3.14159
  - ’a’
  - “Hello World”

  – sind keine *Konstanten*, sondern *Literale*.

Literale

- Wichtig:
  - Eine *Konstante* ist eine Stelle im Hauptspeicher, deren Wert nicht geändert werden kann.
  - Ein *Literal* ist ein explizit ins Quellcode hineingeschriebener und damit natürlich ebenfalls unveränderlicher Wert.
- *Beispiel*:

```
final char var = 'a';
```

Konstante Literal

Objekt- und Klassenkonstanten

- Es existiert analog zu den Objekt- und Klassenvariablen das Konzept der Objekt- und Klassenkonstanten.
- Erinnerung:
  - Die Unterscheidung zwischen Objekt- und Klassenvariablen wurde über das Schlüsselwort **`static`** getroffen.
  - Das Konzept einer Konstanten wurde eingeführt. Diese werden das Schlüsselwort **`final`** deklariert.
- Klassenkonstanten sind:
  - unveränderbar und identisch für alle Objekte

```
static final int stundenlohn = 12;
```

Objekt- und Klassenkonstanten

- Welches sind gültige Variablendeklarationen?
- Beispiel:

```
public class Test{

        public int a = 1;                   // Ok
        public final int b = 1;             // Ok (selten)
        public int c;                       // Ok
        public final int d;                 // Fehler!

        public static int e = 1;            // Ok
        public final static int f = 1;      // Ok
        static public int g;                // Ok
        public static final int h;          // Fehler!

        Konstruktor & Methoden ausgelassen
}
```

Objektvariablen und -konstanten

Klassenvariablen und -konstanten

Objekt- und Klassenkonstanten
Klassen-Konstanten in der Standardbibliothek

- Die Kreiszahl ***`pi = 3,14159`*** ist selbstverständlich reellwertig und konstant (also **`final double`**):
  - **`java.lang.Math.PI`**
- Die wichtigsten Farben sind bereits als Konstanten vom Typ **`java.awt.Color`** mit den entsprechenden RGB–Werten definiert:
  - Klasse **`java.awt.Color`**
    - java.awt.Color.red
    - java.awt.Color.yellow
    - java.awt.Color.green
    - usw.

Objekt- und Klassenkonstanten
Klassen-Konstanten in der Standardbibliothek

- Implementierung der Farbobjekte:

```
public class Color{

        static final Color red = new Color(255, 0, 0);
        static final Color yellow = new Color( 0, 255, 0);
        static final Color green = new Color( 0, 255, 255);

        ...

}
```

  - **`red`** ist logisch gesehen konstant → **`final`**
  - **`red`** ist immer und überall gleich → **`static`**

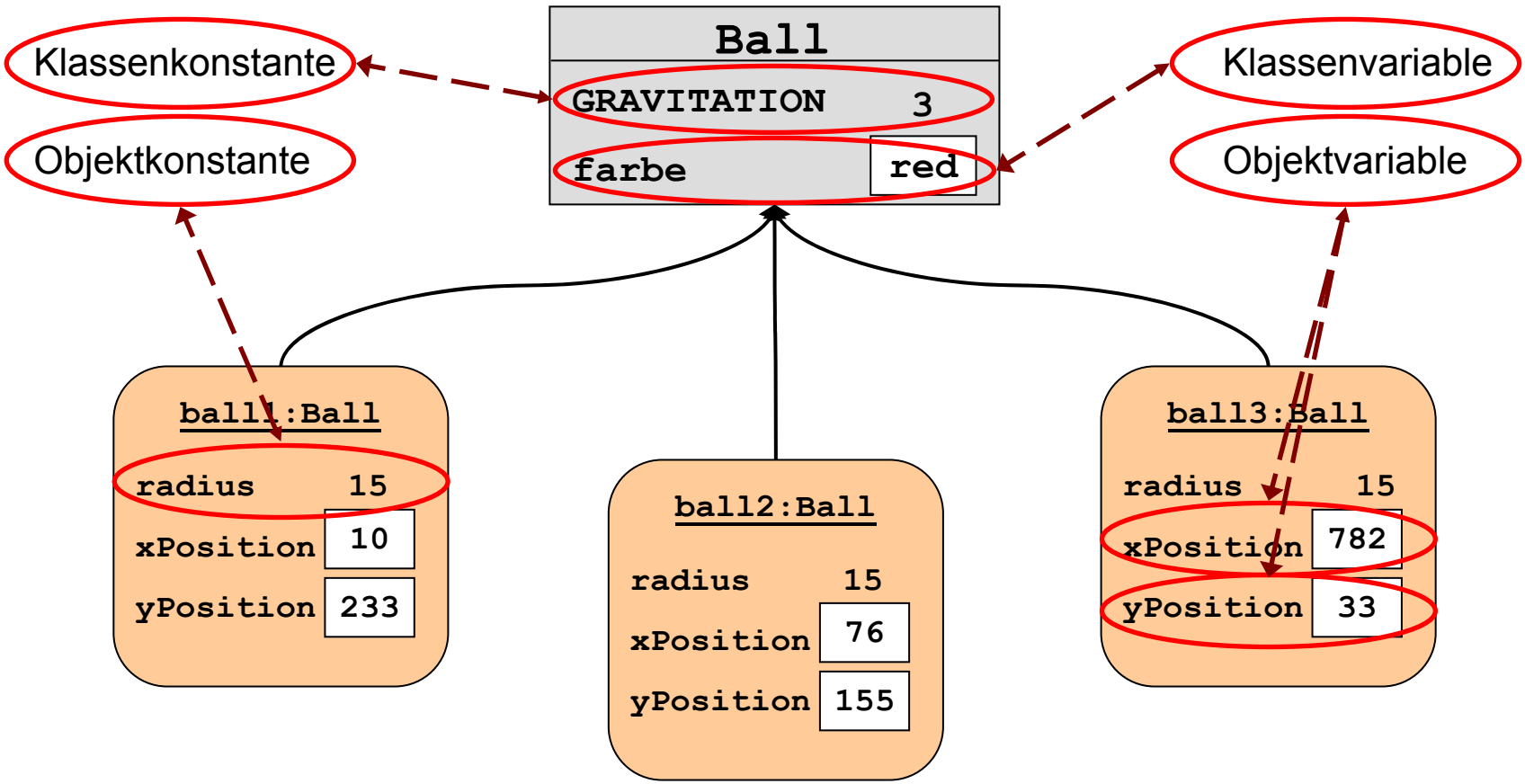
Objekt- und Klassenvariablen
Objekt- und Klassenkonstanten

- abschließendes Beispiel

```
public class Ball {

        private static final int GRAVITATION = 3;
        private static Color farbe = red;
        private final int RADIUS = 15;

        private int xPosition;
        private int yPosition;

        Konstruktor & Methoden ausgelassen
}
```

Objekt- und Klassenvariablen
Objekt- und Klassenkonstanten

- Beispiel



7.2 Gültigkeit von Variablen Scope

- *Erinnerung*: Klammern dürfen immer nur strikt paarweise auftreten.
- *Definition:* Eine Menge von Deklarationen und Anweisungen, die zwischen geschweiften Klammern stehen, wird als ***Block*** bezeichnet.

Scope einer Variablen

Der Scope einer Variable der Bereich von ihrer Deklaration bis zur schließenden Klammer des ersten umschließenden Blocks `{...}`.

Wichtigste Ausnahmen: Variablen können

- über den Kopf von Schleifen und Methoden in einen Block herein-gegeben werden.
- über ein `return`-Statement als Wert aus Blöcken herausgegeben werden.

Scope

- Schematische Beispiele:
  - allgemein

```
{...{... int i; ...{...}...{...{...}...}...}...{...}...}
```

Scope von `i`

  - „for“-Schleife

```
{... for ( int i=0; i<n; i++ ) {...{...}...}...{...}...}
```

Scope von `i`

  - Methode

```
public void f ( int i ) {...{...}...{...}...}
```

Scope von `i`

Scope

- Codebeispiel:

```
public void f (int a) {

  int b = 1;
  if ( a == b ){
     for (int i=0; i<10; i++){
        int c = 2;
     }                          // <- Scope-Ende von c und i
     int d = 3;
     {
       int e = 4;
     }                          // <- Scope-Ende von e
     int f = 5;
     {
       int g = 4;
     }                          // <- Scope-Ende von g
  }                             // <- Scope-Ende von d und f
}                               // <- Scope-Ende von a und b
```

Lebenszeit

- Eine Variable existiert während der Abarbeitung des Blocks in dem sie definiert ist.
- Eine Komponente eines Array- oder Klassenobjektes existiert, solange das Gesamtobjekt existiert.

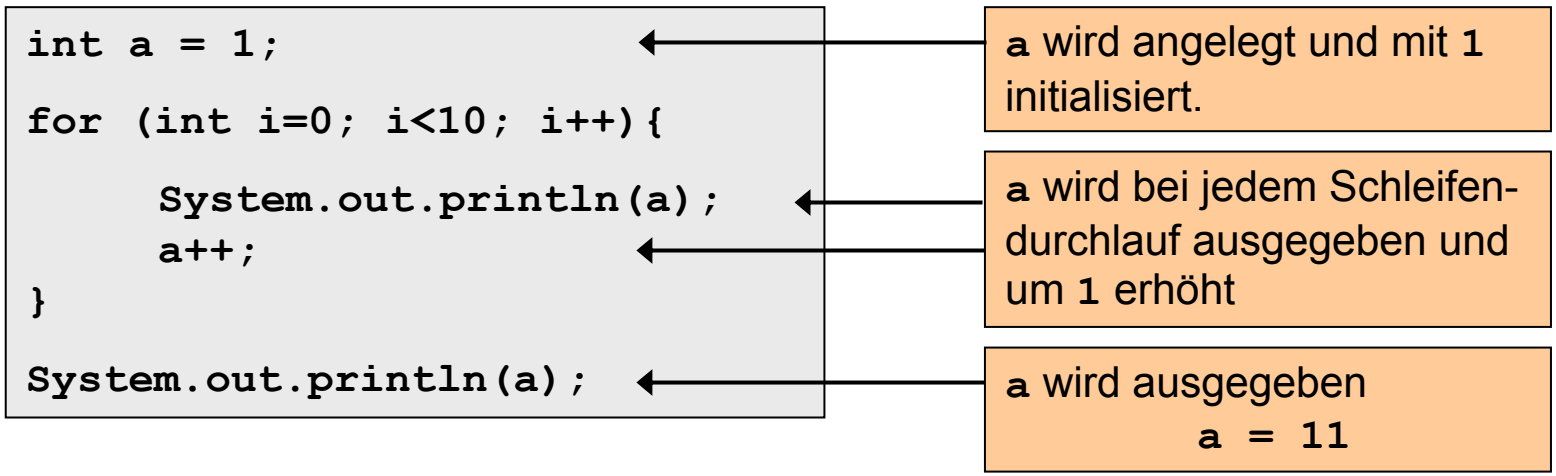
Achtung:

- Wenn der Scope einer Variablen verlassen und wieder betreten wird, wird die Variable
  - nicht nur wieder eingerichtet,
  - sondern auch neu initialisiert.

  → Der alte Wert, den die Variable beim Verlassen des Scopes hatte, ist verloren.
- Das Objekt, auf das eine Variable eines Klassentyps verweist, kann im Speicher durchaus länger als die Variable selbst leben.

Lebenszeit

- allgemeines Beispiel:



Lebenszeit

- Beispiel für eine ständige Variablenneuinitialisierung:

```
for (int i=0; i<10; i++){
      int a = 1;
      System.out.println(a);
      a++;
}

System.out.println(a);
```

`a` wird bei jedem Schleifendurchlauf neu angelegt und mit 1 initialisiert.

`a` wird bei jedem Schleifendurchlauf ausgegeben und um 1 erhöht. (Kein Effekt!)

`a` ist außerhalb des Sopes → **Compiler-Fehler**

Lebenszeit

- Beispiel für Referenzen:

```
public class MeineKlasse{

  public int i;
  public double d;
  public char c;

  Konstruktor & Methoden ausgelassen

}
```

```
for (int i=0; i<10; i++) {

  MeineKlasse a = new MeineKlasse();
  a.i++;
  System.out.println(a.i);

}

System.out.println(a.i);
```

Bei jedem Schleifendurchlauf wird ein Verweis `a` auf ein Objekt von Typ `MeineKlasse` angelegt.

Die Komponenten `a.i`, `a.d` und `a.c` werden *jedes Mal* mit den Standard-Werten initialisiert.

`a` ist außerhalb des Sopes → **Compiler-Fehler**

Lebenszeit

- Beispiel für eine Methode:

```
public void f() {

  MeineKlasse meinVerweis = new MeineKlasse();

     // meinVerweis.i == 0

  System.out.print (meinVerweis.i);
  meinVerweis.i++;
  System.out.print (meinVerweis.i);

}
```

  - Methodenaufruf

```
f();
f();
```

Ausgabe: “01”

Ausgabe: ebenfalls “01” (nicht “12”)

Lebenszeit

- Beispiel für die Rückgabe eines Objekts:

```
public String englischerNikolaus() {
  String str = new String ("Santa Claus");
  return str;
}
```

  - Methodenaufruf

```
String name = englischerNikolaus();
System.out.println (name);
```

Ausgabe: "`Santa Claus`"

  - Erläuterungen
    - Die Zeichenkette **`Santa Claus`** ist über die Variable **`str`** erzeugt worden. Ihre Existenz endet jedoch mit dem Ende der Abarbeitung der Methode **`englischerNikolaus`**.
    - Aber die Zeichenkette **`Santa Claus`** existiert darüber hinaus, weil sie von der Methode als Wert zurückgegeben wird.

Garbage Collection

- Erinnerung:
  - Variablen von Klassen
    - bezeichnen nicht die Objekte selbst,
    - sondern nur *Referenzen* auf die eigentlichen Objekte,
    - und die eigentlichen Objekte müssen mit **`new`** erst noch explizit angelegt werden.
  - Bei der Abarbeitung eines Programms arbeitet im Hintergrund immer das Laufzeitsystem mit.
- Aufgabe des *Laufzeitsystems*:
  - Verwaltung eines ”Pools“ von Speicherplatz.
  - Jedes **`new`** ist eine Anfrage an diese Poolverwaltung.

Garbage Collection

- Abarbeitung von **`new`**:
  - Ein Ausdruck **`new X ...`** liefert als Rückgabe einen Verweis auf ein Objekt der Klasse **`X`** zurück.
    - Falls die Poolverwaltung momentan ausreichend Speicherplatz zur Verfügung hat, wird wie gewünscht ein neues Objekt erzeugt.
    - Ansonsten nach nicht mehr benötigten Objekten suchen und deren Speicherplatz freigeben. Falls nun genug Platz, wie gewünscht neues Objekt erzeugen.
    - Seine Adresse wird als Wert des **`new`**–Ausdrucks zurück geliefert.
    - Dieser Wert kann mit dem Operator **=** einer Variablen der zugehörigen Klasse zugewiesen werden.

Garbage Collection

- Abarbeitung von **`new`** (Fortsetzung):
  - Falls der Speicherplatz hingegen immer noch **nicht** ausreicht, wird ein Fehlermechanismus ausgelöst.
  - Nach momentanem Stand der Vorlesung bedeutet das: unvermeidbarer Programmabsturz.
  - *Später* in Vorlesung und Übungen: Vermeidung des Programmabsturzes durch *Exceptions*.

Garbage Collection

- Speicher wird frei:

```
String str1 = new String ( "Hallo" );
String str2 = str1;
...
str1 = new String ( „Guten Tag" );
str2 = str1;
```

  - Nach Abarbeitung des Codestücks gibt es keinen Verweis auf die zuerst erzeugte Zeichenkette **`Hallo`** mehr.
  - Beide Variablen, die zuerst auf diese Zeichenkette verwiesen haben, sind später auf die Adresse einer anderen Zeichenkette umgesetzt worden.
  - Der für **`Hallo`** reservierte Speicherplatz ist nun vom Programm aus nicht einmal mehr erreichbar.
  - Er steht dem Laufzeitsystem aber nicht für die Bedienung weiterer Anfragen mit **`new`** zur Verfügung.

Garbage Collection

- Speicherüberlauf?

```
String str;
while (true)
  str = new String ("Hallo");
```

- Erläuterung:
  - Mit **`while`** wird bekanntlich eine Schleife eingeleitet, die solange durchlaufen wird, bis die logische Bedingung in Klammern falsch (**`==false`**) wird.
  - Das Literal **`true`** wird natürlich niemals falsch.

    → Endlosschleife.
  - Es wird also endlos neuer Speicherplatz eingerichtet.
- Frage: Ist ein Programmabsturz damit nicht vorprogrammiert?

Garbage Collection

- Mögliche Gegenstrategie
  - Neben **`new`**–Anweisungen gibt es noch ein weiteres Konstrukt, um Speicherplatz wieder an die Poolverwaltung zurückzugeben.
  - Zum Beispiel **`delete`** in C++.
    - Beispielhafter C++–Code:

```
char* str;              // Referenzvariable ("Pointer")
str = new char[100]; // Speicherplatz fuer 100 Zeichen
...
delete [] str;          // Wieder freigegeben
```

  - Ähnliche Konstrukte gibt es in Pascal, C, Ada, ...
  - Aber **nicht** in Java!

Garbage Collection

- weitere Probleme mit einem solchen Konstrukt:
  - In komplexeren Programmen können ein **`new`** und das zugehörige **`delete`** potentiell sehr weit auseinanderliegen,
  - *Praktisch unvermeidliches Resultat*: Schwer zu findende Programmierfehler mit teilweise erheblichen Konsequenzen.
    - Das **`delete`** wird oft vergessen.

      → Wenn oft genug Speicherplatz angefordert, aber nicht zurückgegeben wird, geht irgendwann gar nichts mehr.
    - Ein Stück Speicherplatz, das mit **`delete`** wieder freigegeben (und vielleicht schon weiterverwendet!) wurde, wird aus Versehen weiter benutzt oder ein weiteres Mal mit **`delete`** freigegeben.

  → Programmabsturz ist somit nicht das Schlimmste, was passieren könnte...

Garbage Collection

- Java bedient sich einer anderen Strategie, dem Garbage Collector

- *Ergebnis*: Das Problem ist fast gelöst.
  - Warum nur fast ?
  - Man kann natürlich immer noch zuviel Speicherplatz anlegen, ohne dass ein einziges Byte davon unerreichbar wird. (Beispiel folgt später bei Listen)

Garbage Collector

Das Laufzeitsystem startet hin und wieder im Hintergrund einen zusätzlichen Prozess, der

- alle momentan reservierten Speicherbereiche absucht, und prüft ob sie vom Programm über Referenzen überhaupt noch erreichbar sind.
- jedes als nicht mehr erreichbar klassifizierte Stück Speicherplatz an die Poolverwaltung zurückgibt.

Garbage Collection

- Wie lange existiert ein mit **`new`** erzeugtes Objekt?
  - Das Objekt existiert mindestens noch solange, wie es eine ”Kette“ von Verweisen gibt, über die man das Objekt vom Programm aus ansprechen kann.
  - Wenn die letzte solche Kette abreißt, existiert das Objekt zunächst einmal weiter.
  - Erst wenn der *Garbage Collector* das nächste Mal aktiv wird, vernichtet er das Objekt.
  - Das passiert zu einem Zeitpunkt den der Java–Programmierer weder vorhersehen noch beeinflussen kann.

Kontrollfragen zu diesem Kapitel

1. Warum kann ich beispielsweise auf die Konstante **`java.lang.Math.PI`** direkt zugreifen? Erklären Sie das Konzept der Klassenvariable!

2. In welchem Fall existiert eine Variable außerhalb ihres definierten Blocks?

3. Welche Aufgabe des Java-Laufzeitsystems haben wir kennen gelernt?

4. Warum löst das Konzept der Garbage Collection nicht alle Probleme der Speichernutzung?