Kapitel 07

Variablen und deren Gültigkeit

Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

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

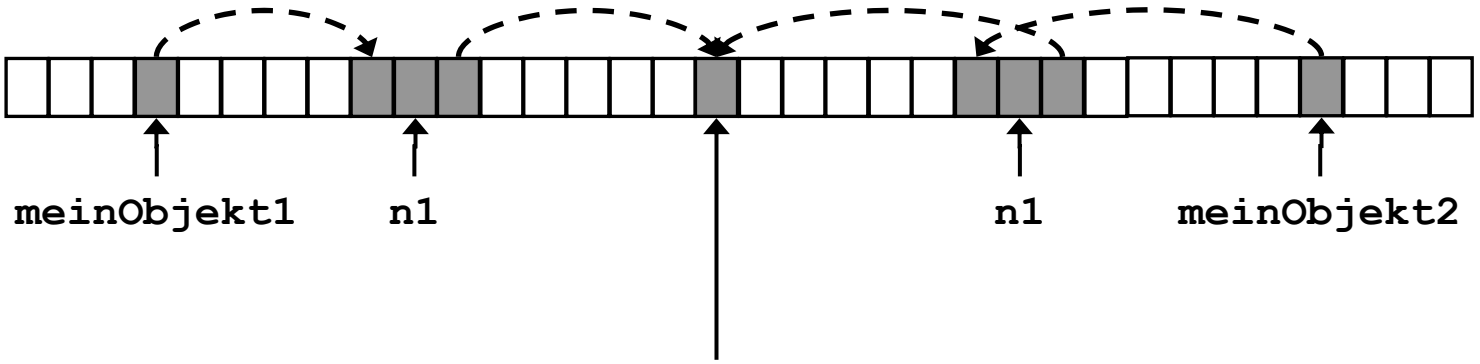
- *Semantische Unterscheidung*:
  - Eine Klassenvariable bzw. –konstante ist ein einzelnes und einmaliges Objekt. Sie werden in der Klasse selber und nicht im Objekt gespeichert.
  - Eine Objektvariable bzw. –konstante gibt es einmal pro erzeugtem Objekt (Instanz) der Klasse.
  - Die Bestandteile von 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,** das ein einziges Mal irgendwo im Speicher angelegt wird und für alle Objekte des Typs **`MeineKlasse`** gilt.

    → **`meinObjekt1.n2`** und **`meinObjekt2.n2`** bezeichnen dasselbe **`int`**–Objekt.

    → 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 einsymbolischer 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).

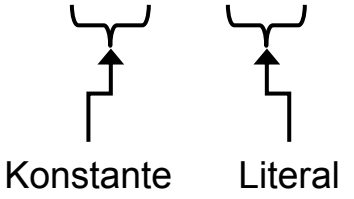
Konstanten und Literale

- *Erinnerung*: Zeichenketten der folgenden Formen
  - 69534
  - 3.14159
  - ’a’
  - “Hello World”
  - sind keine *Konstanten*, sondern *Literale*.

Literale

- Wichtig:
  - Eine *Konstante* ist ein Objekt 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';
```



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
        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`**

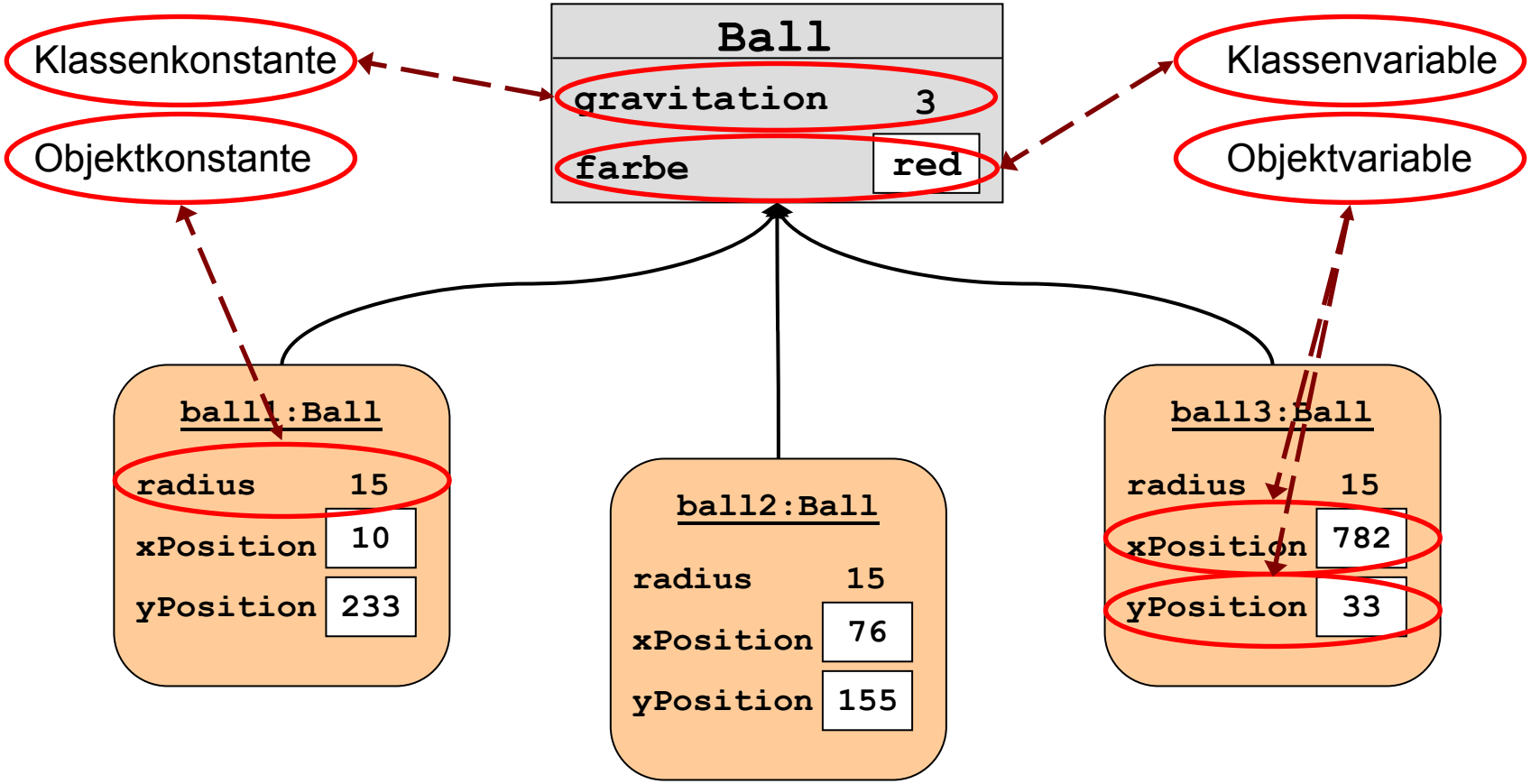
Objekt- und Klassenvariablen
Objekt- und Klassenkonstanten

- 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 hereingegeben 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.

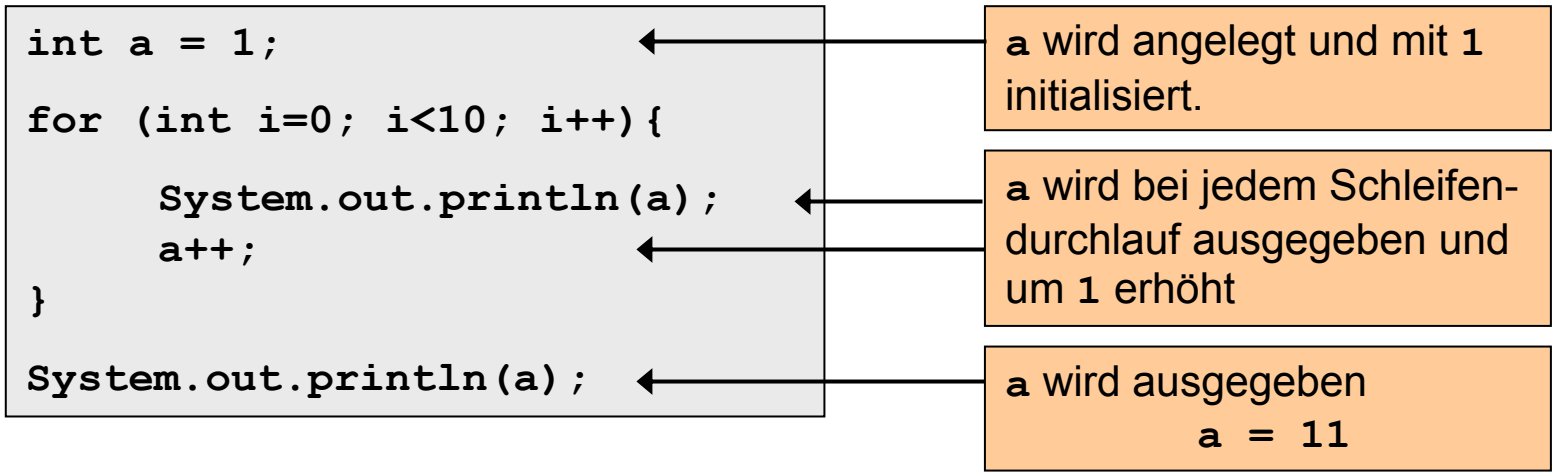
Achtung:

- 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.

Der Garbage Collector

Kurze Erinnerung an früher behandelte Konzepte.

Das Java-Laufzeitsystem übernimmt im Hintergrund Aufgaben, die nicht explizit im Quellcode gestellt wurden, z. B.

- Zugriffsprüfungen
- Abfangen von Ausnahmen wie Division durch 0
- automatische Speicherbereinigung/-verwaltung

Objekte von Klassen werden durch Variablen referenziert.

- Die Objekte selber bestehen aus einer Kombination eingebauter Datentypen und bereits bestehender Klassen.
- Der Aufruf `new (Konstruktor)` erzeugt ein Objekt.

Der Garbage Collector

Was passiert durch einen Aufruf wie im Beispiel unten?

Wenn ausreichend Speicher zur Verfügung steht, wird

- ein Objekt der Klasse `LetterCounter` im Speicher angelegt.
- ein Verweis auf dieses Objekt zurück geliefert.
- dieser Verweis durch das Zuweisungszeichen der Variable `lc` zugewiesen.
  - Die Variable `lc` ist vom Typ `LetterCounter`.

```
1 // Variable lc verweist auf ein Objekt vom Typ LetterCounter
2 LetterCounter lc = new LetterCounter("Zu untersuchender Text");
```

Der Garbage Collector

Was passiert aber, wenn nicht genügend Speicher vorhanden ist?

Es tritt ein Fehler auf.

- Das Java-Laufzeitsystem löst einen Fehlermechanismus aus.
- Der Fehlermechanismus stoppt das Programm, wenn der Fehler nicht individuell behandelt wird.
- Ausblick: Später wird es möglich sein, solche Fehler durch Exceptions zu behandeln, ohne dass das Programm angehalten werden muss.

Der Garbage Collector

Wie kann es überhaupt dazu kommen, dass nicht mehr ausreichend Speicher für die Programmfortsetzung vorhanden ist?

Die Variable `String str1` im Programmstück rechts

- verweist zu erst auf das String-Objekt mit der Zeichenkette `Hallo` und
- verweist anschließend auf ein zweites String-Objekt mit der Zeichenkette `Welt`.

Überschreiben von Referenzen

```
1 // Erzeuge einen String
2 String str1;
3
4 // str1 --> String-Objekt1
5 str1 = new String ("Hallo");
6
7 // str1 --> String-Objekt2
8 str1 = new String ("Welt");
```

Der Verweis in den Variablen verändert sich, die eigentlichen Objekte im Speicher bleiben davon aber unberührt. Sie sind nur vom Programm aus nicht mehr erreichbar.

Der Garbage Collector

Kann man in Java einen Speicherüberlauf erzeugen, der zum Programmabsturz führt?

Eine Schleife erzeugt immer weiter neue String-Objekte.

- Die Bedingung der `while`-Schleife ist immer `true`.
  - Endlosschleife
- Es wird also endlos weiterer Speicherplatz durch neue Objekte belegt.
- Frage: Ist ein Programmabsturz damit vorprogrammiert?

Endlosschleife

```
1 String str;
2
3 while (true){
4   str = new String("Hallo");
5   System.out.prinln(str);
6 }
```

Der Garbage Collector

Mögliche Gegenstrategien zur Freigabe von belegtem, aber unbenutztem Speicher.

Eine Art inverse Anweisung zu `new`.

- C++ stellt eine Funktion `delete` zur Verfügung.
- Mit dieser Anweisung kann belegter Speicher wieder frei gegeben werden.
- Ähnliche Konstrukte gibt es für Pascal, C, Ada u. a. Programmiersprachen.

Der Garbage Collector

Mögliche Gegenstrategien zur Freigabe von belegtem, aber unbenutztem Speicher.

Nachteile der Funktion `delete` als eine Art inverse Anweisung zu `new`.

- In komplexen Programmen können ein `new` und das zugehörige `delete` potentiell sehr weit von einander entfernt liegen.
- Praktisch unvermeidliches Resultat: Schwer zu findende Programmierfehler mit teilweise erheblichen Konsequenzen.
  - Das `delete` wird oft vergessen und es kommt zum Speicherüberlauf: Programmabsturz.
  - Ein Stück Speicherplatz, das mit `delete` freigegeben wurde, wird an späterer Stelle weiter benutzt oder ein weiteres Mal mit `delete` freigegeben: Schäden am Ergebnis, Programm oder System.

Der Garbage Collector

Javas Gegenstrategie ist der Einsatz des Garbage Collectors.

- Der Garbage Collector gibt verwendeten Speicher automatisch wieder frei, auf den keine Referenzen mehr existieren.
- Ergebnis: Das Problem ist fast gelöst.
  - Warum nur fast ?
  - Man kann auch unendlich viele Objekte anlegen, ohne die Referenzen zu überschreiben.
  - Garbage Collector arbeitet automatisch, evtl. auch zu spät.

Endlosschleife

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

- alle momentan reservierten Speicherbereiche absucht,
- prüft ob sie vom Programm über Referenzen noch erreichbar sind und
- alle nicht mehr erreichbaren Speicherbereiche wieder frei gibt.

Der Garbage Collector

Wie lange existiert nun ein mit new erzeugtes Objekt in Java?

- 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 Kapitel 07

1. Warum kann beispielsweise auf die Konstante `java.lang.Math.PI` direkt zugegriffen werden? Erklären Sie das Konzept der Klassenvariablen!

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

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

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