- sie dürfen keinesfalls zu Attributen gemacht werden, nur „weil es sich dann einfacher programmieren lässt“
  - es entstünden sonst dieselben Probleme wie wenn man in der prozeduralen Programmierung (z.B. mit C) statt lokaler Variablen globale Variablen benutzen würde

- in Java gibt es (fast) alle Anweisungen, die auch aus C bekannt sind:
  - Ausdrucksanweisungen
  - die leere Anweisung
  - Blockanweisungen
  - `if`-Anweisung

- `switch`-Anweisung
- `do-while`-Anweisung
- `while`-Anweisung
- `for`-Anweisung
- `return`-Anweisung
- `break`-Anweisung
- `continue`-Anweisung
- `assert`-Anweisung

- sie verhalten sich in Java im wesentlichen genauso, wie in C

- darüber hinaus gibt es vier weitere Anweisungen, die wir zu gegebener Zeit kennen lernen werden:
  - `for-each`-Anweisung
  - `throw`-Anweisung
  - `try`-Anweisung
  - `synchronized`-Anweisung

- die in C/C++ vorhandene `goto`-Anweisung gibt es in Java nicht

- jetzt sind wir (endlich) in der Lage, uns der Deklaration der Konstruktoren und der Methoden unserer Klasse `AnnuitaetenDarlehen` zu widmen

- welchen Zugriffsmodifizierer sollen wir für die Konstruktoren wählen?
  - würden wir `private` wählen, könnten die Konstruktoren außerhalb der Klasse nicht benutzt werden
  - weil aber zur Erzeugung von Objekten immer Konstruktoren erforderlich sind, wäre die Folge, dass außerhalb der Klasse keine Objekte erzeugt werden könnten

- wir hätten dann einen „Bauplan für Experten“, deren Fähigkeiten (nahezu) niemand nutzen könnte!

- **Fazit:** Konstruktoren sind in der Regel **`public`**
  - wie wir noch sehen werden, gibt es aber auch Ausnahmen

- wir ersetzen den im UML-Entwurf vorgesehenen Datentyp **`Datum`** durch die Klasse **`GregorianCalendar`** der Java-Klassenbibliothek und deklarieren die Köpfe unserer Konstruktoren so:


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class AnnuitaetenDarlehen
{
  private double betrag;
  private int laufzeit;
  private double zinssatz;
  private GregorianCalendar startdatum;

  public AnnuitaetenDarlehen(double derBetrag, int dieLaufzeit,
                             double derZinssatz)
  {
    // Anweisungen
  }

  public AnnuitaetenDarlehen(double derBetrag, int dieLaufzeit,
          double derZinssatz, final GregorianCalendar dasStartdatum)
  {
    // Anweisungen
  }

  // Methoden
}
```


- zur Gestaltung der Rümpfe der Konstruktoren müssen wir zunächst folgende Fragen klären
  - wie werden in Java Objekte erzeugt?
    - denn im ersten Konstruktor müssen wir ein Objekt der Klasse **`GregorianCalendar`** erzeugen, das den Zeitpunkt „heute“ repräsentiert
  - was sollen wir machen, wenn der Input für die Parameter **`derBetrag`**, **`dieLaufzeit`**, **`derZinssatz`** oder **`dasStartdatum`** fehlerhaft ist?
    - denn wir müssen sicherstellen, dass mit den von uns gestalteten Konstruktoren nur sinnvolle „Darlehensexperten“ erzeugt werden können

- der für objektorientierte Sprachen typische Mechanismus zum Erzeugen von Objekten ist in Java mit Hilfe des unären Operators **`new`** realisiert
  - im einfachsten Fall ist der Operand des **`new`**-Operators ein Konstruktor der Klasse, nach deren Bauplan das Objekt erzeugt werden soll
  - dabei müssen dem Konstruktor gegebenenfalls Inputwerte übergeben werden:

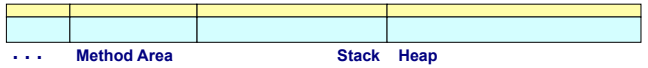
```
GregorianCalendar datum = new GregorianCalendar(2010, 11, 16);
```

- es gibt es aber auch Varianten des `new`-Operators, die erheblich komplexer sind

- was bewirkt der `new`-Operator?

- um das zu verstehen, betrachten wir zunächst, wie die JVM den ihr zur Ausführung eines Java-Programms (vom Betriebssystem) zur Verfügung gestellten Speicherplatz nutzt
- sie teilt ihn in mehrere Bereiche (Segmente) ein, die jeweils einem bestimmten Zweck dienen

- die für uns wichtigsten Segmente sind
  - die Method Area
  - der Stack
  - der Heap



- in der Method Area legt die JVM für jede der im Programm benutzten Klassen ab
  - den (Binär-)Code der Anweisungen der Methoden und Konstruktoren

- Verwaltungsinformationen, die benötigt werden, um auf die Attribute und Methoden der Klasse zugreifen zu können (Runtime Constant Pool)
- sogenannte Native Method Stacks
  - dies aber nur dann, wenn in den Methoden Code eingebunden wurde, der in einer anderen Programmiersprache geschrieben wurde (z.B. in C)

- die Größe der Method Area wird initial von der JVM festgelegt
  - bei Bedarf kann sie von der JVM vergrößert werden

- wann immer eine Methode aktiviert wird, werden deren lokale Daten (in einem sogenannten Stack Frame) auf dem Stack gespeichert
  - das sind vor allem
    - die Parameter und die lokalen Variablen der Methode
    - die Rücksprungadresse, an der das Programm fortgesetzt werden soll, wenn die Ausführung der Methode beendet ist
  - außerdem wird bei Bedarf Speicherplatz allokiert für
    - Zwischenergebnisse, die bei der Auswertung komplexer Ausdrücke anfallen können
    - den Rückgabewert der Methode
    - Referenzen auf weiterzureichende Exceptions

- sobald die Ausführung einer Methode beendet ist, entfernt die JVM den Stack Frame der Methode aus dem Stack
  - genauer: die JVM überschreibt den vom Stack Frame belegten Speicherplatz (so weit wie nötig) durch den Stack Frame der nächsten auszuführenden Methode
- wird die Methode später erneut aktiviert, wird für sie wieder ein neuer Stack Frame auf dem Stack angelegt
- **Achtung:** dies triff auch dann zu, wenn eine Methode sich selbst aufruft

  - eine derartige Methode wird direkt rekursiv genannt
  - ruft sie sich über mehrere Zwischenschritte selbst auf, handelt es sich um eine indirekt rekursive Methode
  - wird bei einer rekursiven Methode das Anlegen von Stack Frames nicht unterbrochen, werden so lange neue Stack Frames angelegt, bis der zur Verfügung gestellte Speicherplatz aufgebraucht ist und das Programm mit einem Stack Overflow abbricht
  - **Fazit:** rekursive Methoden müssen in der Regel eine Abbruchbedingung enthalten, die das ständige Anlegen von Stack Frames rechtzeitig beendet

- auf dem Stack werden also ständig Daten geschrieben und wieder „gelöscht“
- sofern es sich dabei um Daten handelt, die einen primitiven Datentyp haben, ist der damit verbundene Aufwand vorhersehbar und wegen der geringen Größe der Werte akzeptabel (jeweils 1 bis 8 Byte)
- bei Referenzdatentypen ist die Situation anders
  - Objekte können im Prinzip beliebig groß sein
  - das ständige Erzeugen, Löschen und wieder neu Erzeugen von Objekten könnte daher unkalkulierbar viel Zeit in Anspruch nehmen

- die JVM behandelt daher die „Werte“ von Referenzdatentypen (= Objekte) anders als die Werte primitiver Datentypen
  - Objekte werden immer auf dem Heap gespeichert
  - einmal erzeugt, verbleiben sie dort in der Regel so lange, bis das Programm beendet ist
    - da der Heap ziemlich groß ist, entfernt die JVM nur bei sehr speicherintensiven Programmen gelegentlich Objekte vom Heap
  - auf dem Stack werden lediglich Referenzen (= Zeiger) auf die Objekte gespeichert
    - diese sind klein und haben eine konstante Größe (4 Byte)

- die Intialisierung

```
GregorianCalendar datum = new GregorianCalendar(2010, 11, 16);
```

bewirkt also im wesentlichen Folgendes:

- der **new**-Operator allokiert auf dem Heap so viel Platz, wie zur Speicherung (der Attribute) eines Objektes der Klasse **GregorianCalendar** nötig ist
- dann führt er zur Initialisierung des Speicherbereichs (= Objektes) seinen Operanden aus, d.h. den Konstruktor der Klasse **GregorianCalendar**
- das Ergebnis der Operation ist die Anfangsadresse des auf dem Heap allokierten Speicherbereichs, d.h. eine Referenz (= Zeiger) auf das Objekt

- diese Adresse wird **datum** zugewiesen

- bei **datum** muss es sich daher um eine Referenz, d.h. um einen Zeiger, handeln
- wie in C/C++, veranschaulichen wir uns die Situation so:

- **Achtung:** anders als in C/C++ ist die Tatsache, dass es sich bei **datum** um eine Referenz handelt, in Java nicht an Hand einer speziellen Notation erkennbar

- in Java sind alle Werte (z.B. Attribute, Parameter, lokale Variablen, Rückgabewerte von Methoden), deren Datentyp eine Referenzdatentyp ist (d.h. eine Klasse, ein **enum**, ein Array oder ein Interface) in Wirklichkeit Referenzen (= Zeiger) auf Objekte, die sich auf dem Heap befinden, oder Referenzen auf **null**
  - durch

```
GregorianCalendar datum;
```

wird also kein Objekt deklariert, sondern nur eine Referenz (= Zeiger) auf ein noch in einem zweiten Schritt mit Hilfe von **new** zu erzeugendes Objekt

- wie können wir die Fähigkeiten von Objekten nutzen, die mit Hilfe von **new** erzeugt wurden?
  - indem wir sie nach folgendem Muster beauftragen, ihre Methoden auszuführen:

```
objekt.machWas()
```

  - konkret sieht das z.B. so aus:

```
boolean schaltjahr = datum.isLeapYear(2000);
```

    - hierbei ist **isLeapYear** eine **public**-Methode, die im „Bauplan“ **GregorianCalendar** festgelegt wurde und daher im Objekt **datum** zur Verfügung steht

      genauer: in dem Objekt, auf das die Referenz **datum** zeigt

- nach diesen Erkenntnissen können wir uns der zweiten Frage zuwenden:
- was sollen wir bei unseren Konstruktoren machen, wenn der Input für die Parameter **`derBetrag`**, **`dieLaufzeit`**, **`derZinssatz`** oder **`dasStartdatum`** fehlerhaft ist?
- üblich sind folgende Vorgehensweisen:
  - der unsinnige Input wird ignoriert und stattdessen ein Objekt mit zulässigen Default-Werten erzeugt
  - es wird dafür gesorgt, dass das Programm nicht weiter ausgeführt werden kann

- **Achtung:** in beiden Fällen ist in der Dokumentation der Klasse genau anzugeben, wie sich die Konstruktoren bei falschem Input verhalten
- wir entscheiden uns für den zweiten Weg
- es gibt mehrere Möglichkeiten, ihn zu realisieren
  - am sinnvollsten ist es, eine so genannte Exception zu erzeugen und mit Hilfe der **`throw`**-Anweisung zu „werfen“
  - wird die Exception nicht „gefangen“ und die Fehlersituation adäquat behandelt, bricht die JVM die Ausführung des Threads ab, d.h. bei uns das Programm

- was sind Exceptions?
  - inhaltlich: „Experten zum Signalisieren von Fehlern“
  - formal: Objekte einer Exception-Klasse
- die Java-Klassenbibliothek enthält für verschiedene Fehlersituationen mehr als 360 Exception-Klassen, die über diverse Packages verteilt sind
  - darüber hinaus können (beliebig viele) eigene Exception-Klassen geschrieben werden
- weil Exceptions Objekte sind, müssen auch sie mit Hilfe des **`new`**-Operators erzeugt werden

- Exceptions können automatisch von der JVM erzeugt und „geworfen“ werden
  - wenn sie einen so genannten Laufzeitfehler erkennt
- Exceptions können aber auch durch Konstruktoren und Methoden erzeugt und mit Hilfe der **`throw`**-Anweisung „geworfen“ werden
  - davon machen wir Gebrauch
- die **`throw`**-Anweisung ist nach diesem Schema zu gestalten:

```
throw Ausdruck;
```

Achtung: der Wert von **`ausdruck`** muss eine Referenz auf ein Objekt einer Exception-Klasse sein

- „wirft“ ein Konstruktor oder eine Methode mit Hilfe der **`throw`**-Anweisung eine Exception, wird die „normale“ Ausführung sofort unterbrochen und eine der beiden folgenden Situationen tritt ein
  - die „geworfene“ Exception wird innerhalb des Konstruktors oder der Methode durch eine **`catch`**-Klausel einer **`try`**-Anweisung „gefangen“, um die Fehlersituation adäquat zu behandeln
  - die „geworfene“ Exception wird innerhalb des Konstruktors oder der Methode nicht „gefangen“

- im zweiten Fall wird die „geworfene“ Exception an die Aufrufer des Konstruktors oder der Methode „weitergereicht“
  - die können dann die Exception „fangen“, um die Fehlersituation zu behandeln
  - sie können sie aber auch an ihre Aufrufer einfach „weiterreichen“
  - usw.
- wird die „geworfene“ Exception nie „gefangen“, bricht die JVM die Ausführung des Threads ab, d.h. bei uns bisher das Programm

- jetzt verstehen wir auch den Zweck der **`throws`**-Klausel von Konstruktoren oder Methoden:
- reicht ein Konstruktor oder eine Methode Exceptions an Aufrufer weiter, sollten diese das wissen
  - „damit sie sich darauf einstellen können“
- die **`throws`**-Klausel dient dazu, den Aufrufern mitzuteilen, womit sie rechnen müssen, d.h. welche Exceptions weiter gereicht werden
  - hinter dem Schlüsselwort **`throws`** ist anzugeben, von welchem Typ die weitergereichten Exceptions sind, d.h. der Name der zugehörigen Exception-Klassen

- das folgende Beispiel zeigt, dass es dabei unerheblich ist, ob die weitergereichte Exception durch eine **`throw`**-Anweisung des Konstruktors oder der Methode selbst „geworfen“ wurde oder von einer der benutzten Methoden erzeugt wurde und einfach nur weitergereicht wird:

```
public String erfasseText() throws IOException
{
  InputStreamReader ausTastatur = new InputStreamReader(System.in);
  BufferedReader eingabe = new BufferedReader(ausTastatur);
  return eingabe.readLine();
}
```

- die Methode **`readLine`** der Klasse **`java.io.BufferedReader`** reicht im Fehlerfall eine **`IOException`** weiter
  - das kann man ihrer Dokumentation entnehmen
- weil diese in **`erfasseText`** nicht gefangen wird, ist die **`throws`**-Klausel erforderlich

- reicht eine Methode mehrere Exceptions unterschiedlicher Typen weiter, sind in der **`throws`**-Klausel – durch Kommata getrennt – alle Typen anzugeben
  - dabei kommt es nicht auf die Reihenfolge an

Beispiel:

```
private void holeDaten() throws FileNotFoundException, EOFException
{
  // Anweisungen
}
```

- **Achtung:** in der **`throws`**-Klausel müssen nur so genannte checked Exceptions angegeben werden
  - Exceptions, die unchecked sind, werden nicht aufgeführt
  - den Unterschied zwischen checked Exceptions und unchecked Exceptions lernen wir zu gegebener Zeit kennen

- für die Konstruktoren unserer Klasse **`AnnuitaetenDarlehen`** sind Exceptions vom Typ **`java.lang.IllegalArgumentException`** geeignet
  - sie werden eingesetzt, um zu signalisieren, dass Input-Parametern von Konstruktoren oder Methoden falsche Werte übergeben wurden

- weil es sich um unchecked Exceptions handelt, haben die Konstruktoren unserer Klasse keine **`throws`**-Klausel


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class AnnuitaetenDarlehen
{
  private double betrag;
  private int laufzeit;
  private double zinssatz;
  private GregorianCalendar startdatum;

  public AnnuitaetenDarlehen(double derBetrag,
                             int dieLaufzeit,
                             double derZinssatz)
  {
    boolean inputOk = (1.0 <= derBetrag && derBetrag <= 1000000000.0) &&
                      (1 <= dieLaufzeit && dieLaufzeit <= 100) &&
                      (0.1 <= derZinssatz && derZinssatz <= 30.0);
    if (inputOk)
    {
      betrag = derBetrag;
      laufzeit = dieLaufzeit;
      zinssatz = derZinssatz;
      startdatum = new GregorianCalendar();              // heutiges Datum
    }
    else
      throw new IllegalArgumentException();
  }
```

```
public AnnuitaetenDarlehen(double derBetrag,
                           int dieLaufzeit,
                           double derZinssatz,
                           final GregorianCalendar dasStartdatum)
{
  boolean inputOk = (1.0 <= derBetrag && derBetrag <= 1000000000.0) &&
                    (1 <= dieLaufzeit && dieLaufzeit <= 100) &&
                    (0.1 <= derZinssatz && derZinssatz <= 30.0) &&
                    (dasStartdatum != null);
  if (inputOk)
  {
    betrag = derBetrag;
    laufzeit = dieLaufzeit;
    zinssatz = derZinssatz;
    startdatum = dasStartdatum;            // als Input übergebenes Datum
  }
  else
    throw new IllegalArgumentException();
}

// Methoden
}
```


- **Achtung:** wenn eine Klasse mehrere Konstruktoren haben soll, müssen sich diese in der Signatur unterscheiden
  - die Signatur eines Konstruktors oder einer Methode ist die Kombination bestehend aus dem Namen und der in Klammern stehenden Parameterliste
    - dabei kommt es nur auf die Reihenfolge der Datentypen der Parameter an, nicht auf deren Namen
  - unsere Konstruktoren haben die Signaturen

    `AnnuitaetenDarlehen(double, int, double)`

    und

    `AnnuitaetenDarlehen(double, int, double, GregorianCalendar)`

- hat eine Klasse mehrere Konstruktoren, sagt man, dass der Konstruktor der Klasse überladen wurde (overloading)
  - es ist typisch, das Konstruktoren von Klassen überladen werden
- wird kein Konstruktor deklariert, erzeugt die JVM automatisch einen so genannten Default-Konstruktor
  - der initialisiert die Attribute der Klasse mit ihren Default-Werten und hat daher keine Parameter
  - das wäre für unsere Klasse unangemessen

- widmen wir uns nun den Methoden unserer Klasse **`AnnuitaetenDarlehen`**
- für die Gestaltung von Methoden sind folgende Überlegungen von fundamentaler Bedeutung
  - Objekte haben einen Zustand, der durch die Werte ihrer Attribute beschrieben wird
  - manche Zustände, d.h. Belegungen der Attribute, sind aus inhaltlicher Sicht sinnvoll, andere nicht
  - Objekte werden beim Erzeugen (durch den Konstruktor) immer in einen sinnvollen Initialzustand gebracht

- Methoden stellen Fähigkeiten dar, die in sich abgeschlossen sind und unabhängig von anderen Fähigkeiten genutzt werden können
- soll eine Fähigkeit zur allgemeinen Nutzung zur Verfügung stehen, muss die entsprechende Methode mit dem Zugriffsmodifizierer `public` versehen werden
- Methoden haben uneingeschränkten Zugriff auf die Attribute, d.h. (nur) sie können den Zustand der Objekte verändern
- dabei ist darauf zu achten, dass Methoden, die `public` sind, die Objekte nie in einen Zustand bringen, der aus inhaltlicher Sicht unsinnig ist

- **Fazit:** Methoden, die `public` sind, müssen Objekte immer von einem sinnvollen Zustand in einen sinnvollen Zustand überführen und in beliebiger Reihenfolge ausgeführt werden können
- man kann sich das Prinzip gut an Hand einer Tastatur veranschaulichen:
  - jede Taste stößt eine Fähigkeit an (das Einstellen des Zeichens in den Eingabepuffer)
  - die Tasten können in beliebiger Reihenfolge betätigt werden
  - bei Nutzung in bestimmter Reihenfolge entsteht etwas Sinnvolles (zum Beispiel ein Java-Programm)

- wichtig ist, dass es den Nutzern überlassen bleibt, die Methoden von Objekten in einer Reihenfolge zu aktivieren, die für sie von Nutzen ist !
  - unterschiedliche Zwecke erfordern im allgemeinen unterschiedliche Reihenfolgen
- die Methoden einer Klasse, die `public` sind, bilden das so genannte API (Application Programming Interface) der Klasse
- Methoden, die nur internen Zwecken dienen, sollen als `private` deklariert werden

- bei ihnen handelt es sich um Hilfsmethoden, deren Existenz nach außen verborgen werden soll
- sie müssen nicht unbedingt in beliebiger Reihenfolge ausgeführt werden können
- es ist auch nicht zwingend, dass sie ihre Objekte von einen sinnvollen Zustand in einen sinnvollen Zustand überführen

- die Trennung in ein nach außen sichtbares API (öffentliche Methoden) und in unsichtbare Interna (private Attribute und Methoden) hat den Vorteil, dass Interna jederzeit geändert werden können, sofern das bereits vorhandene API davon nicht betroffen ist

- das ermöglicht nachträgliche Optimierungen (weniger Speicherplatz, besseres Laufzeitverhalten, etc.)
- außerdem kann das API jederzeit erweitert werden
  - die „Experten“ können danach einfach mehr

- bei der Gestaltung von Methoden des API, die den (sinnvollen) Zustand der Objekte nicht verändern, ist lediglich darauf zu achten, dass sie – unabhängig von anderen Methoden – jederzeit benutzt werden können
  - sie sind in diesem Sinne „harmlos“

  Hinweis: in C++ können sie speziell gekennzeichnet werden, in Java ist das nicht vorgesehen

- das trifft z. B. für folgende Methoden unserer Klasse **`AnnuitaetenDarlehen`** zu:

```
public double liefereDarlehensBetrag()
{
  return betrag;
}
```

```
public int liefereLaufzeit()
{
  return laufzeit;
}
```

```
public double liefereZinssatz()
{
  return zinssatz;
}
```


- problematischer ist die Gestaltung der Methode **`liefereAuszahlungsDatum`**
- es ist verlockend, sie analog einfach so zu deklarieren:

```
public GregorianCalendar liefereAuszahlungsDatum()
{
  return startdatum;                          // nein !!!
}
```

- das wäre aber ein schwerer Fehler !
- um das einzusehen, erinnern wir uns daran, dass **`startdatum`** lediglich eine Referenz (= Zeiger) auf ein Objekt auf dem Heap ist

- die Anweisung

```
return startdatum;
```

würde lediglich die in **`startdatum`** gespeicherte Adresse einer anderen Referenz, zum Beispiel **`auszahlungsdatum`**, als Wert zuweisen, wodurch diese Situation entstünde: