Klassen und Objekte

Retrospektive

- Für die Modellierung von Dingen oder Sachverhalten nutzt man in c Strukturen.
- Strukturen sind dabei benutzerdefinierte Datentypen, die Komponenten verschiedenen Datentypes zu einem neuen Datentyp vereinen.
- Von diesem Datentyp können (beliebig) viele Objekte erzeugt werden, die dann jeweils ein Ding oder einen konkreten Sachverhalt modellieren.

- In diesem Sinn wird ein konkreter Student, z.B. Hans Huckebein, durch ein Objekt zugegebenermaßen abstrahiert beschrieben.
- Alle Objekte, die in dieser Weise Studenten beschreiben, werden ihrerseits durch den Dytentyp tStudent beschrieben.
- Die Deklaration eines solchen benutzerdefinierten Datentyps erfolgt in den meisten Fällen in einem Headerfile, in dem auch die Prototypen der Funktionen, die zur Arbeit mit eben diesem Datentyp bereitgestellt werden, angegeben sind.

Als Beispiel ein struct btime

```
#ifndef _H_BTIME_
#define _H_BTIME_

typedef struct
{
  int h;
  int m;
}btime;

btime btimeGetCurrentTime();
void btimeShow(btime* pt);
btime btimeIncrement(btime t);

#endif
```

Datenstruktur

Funktionsprototypen dazu

Datei,
gespeichert in btime.h

- Die Struktur btime dient der Modellierung einer Uhrzeit, bestehend aus Stunde(h) und Minute(m).
- Es gibt beispielhaft drei Funktionen zur Arbeit mit dem „Zeit-struct“.
- btime btimeGetCurrentTime();
  - Besorgt die aktuelle Systemzeit und trägt davon die Stunde und die Minute in eine neue Variable unserer Struktur und gibt diese als Wert zurück.
- void btimeShow(btime* pt);
  - Gibt die Werte als Zeit im Format hh:mm aus
- btime btimeIncrement(btime t);
  - Berechnet die Zeit der nächsten Minute

- Implementiert sind die drei Funktionen in der c-Datei btime.c wie folgt:

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

btime btimeGetCurrentTime()
{
  btime tmp={};
  time_t lc = time(NULL);
  struct tm*plc = localtime(&lc);
  int h,m;
  tmp.h=plc->tm_hour;
  tmp.m=plc->tm_min;
  return tmp;
}
```

```
void btimeShow(btime* pt)
{
  printf("%02d:%02d",
                    pt->h,pt->m);
}

btime btimeIncrement(btime t)
{
  t.m++;
  if (t.m==60)
    {t.m = 0; t.h++;}
  if (t.h==24)
    {t.h = t.m = 0;}
  return t;
}
```

- Verwendung findet unser keiner btime – Modul in der c-Datei btimeMain.c

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

int main(int argc, char*argv[])
{
  btime bt={};
  if (argc==3)
    {bt.h=atoi(argv[1]); bt.m=atoi(argv[2]);}
  else bt=btimeGetCurrentTime();
  printf("time is now  ");
  btimeShow(&bt);
  putchar('\n');
  bt=btimeIncrement(bt);
  printf("next time is ");
  btimeShow(&bt);
  putchar('\n');
}
```

- Das Ganze können wir nun mit

  gcc btime.c btimemain.c
- Oder

  gcc btime*.c
- Nach a.out compilieren und linken.

```
./a.out
time is now  18:04
next time is 18:05
beck@U330p:~/SS2016/cpp/class$ ./a.out 19 00
time is now  19:00
next time is 19:01
```

Zusammenfassend ist festzustellen:

- Es gibt einen Programmmodul btime bestehend aus
  - Einer öffentlichen Schnittstelle, die die Struktur und die dazugehörige Funktionalität in Form der Funktionen beschreibt (btime.h)
  - Einer Implementationsdatei btime.c, die die Funktionalität implementiert. Dieser Teil könnte auch in Form einer vorkompilierten Datei btime.o oder als Library libbitime.a oder Teil einer Library vorliegen.
  - Die, im Headerfile enthaltene Struktur kann ggf. vor dem Benutzer auch noch verborgen werden, so wäre der Nutzer definitiv auf die öffentlichen Funktionen angewiesen, um mit dem Modul zu arbeiten.

Geändertes Beispiel mit verborgenem struct Typ

btimep.h

```
#ifndef _H_BTIME_
#define _H_BTIME_

struct btimes;
typedef struct btimes btime;

btime* btimeCreate();
void btimeSetH(btime*pbt, int h);
void btimeSetM(btime*pbt, int m);
void btimeGetCurrentTime(btime*);
void btimeShow(btime* pt);
void btimeIncrement(btime* t);

#endif
```

Leere struct-foreward deklaration
Allerdings kann der Benutzer
nun keine Variable, nur Pointer
des struct-Types anlegen!

btimep.c

```
#include <stdio.h>
#include <time.h>
#include <malloc.h>
#include "btimep.h"

typedef struct
{
  int h;
  int m;
}btime;

void btimeSetH(btime*pbt, int h)
   {pbt->h=h;}
void btimeSetM(btime*pbt, int m)
   {pbt->m=m;}

btime* btimeCreate()
{
  btime*pbt=malloc(sizeof (btime));
  pbt->h=pbt->m=0;
  return pbt;
}
```

Hier ist jetzt die vollständige Strukturdeklaration

```
void btimeGetCurrentTime(btime*bt)
{

  time_t lc = time(NULL);
  struct tm*plc = localtime(&lc);
  int h,m;
  bt->h=plc->tm_hour;
  bt->m=plc->tm_min;
}

void btimeShow(btime* pt)
{
  printf("%02d:%02d",pt->h,pt->m);
}

void btimeIncrement(btime* t)
{
  t->m++;
  if (t->m==60)
    {t->m = 0; t->h++;}
  if (t->h==24)
    {t->h = t->m = 0;}
}
```

```
#include <stdio.h>
#include <time.h>
#include "btimep.h"

int main(int argc, char*argv[])
{
  btime *pbt;
  pbt=btimeCreate();
  if (argc==3)
   {
      btimeSetH(pbt,atoi(argv[1]));
      btimeSetM(pbt,atoi(argv[2]));
   }
  else btimeGetCurrentTime(pbt);
  printf("time is now  ");
  btimeShow(pbt);
  putchar('\n');
  btimeIncrement(pbt);
  printf("next time is ");
  btimeShow(pbt);
  putchar('\n');
}
```

Vom c-struct zur Klasse

- Eine Klasse dient in gleicher Weise, wie ein struct-Datentyp der Modellierung.
- Betrachtet man die Funktionen zum c-Struct genauer, stellt man fest, dass alle einen Parameter von dem struct-Typ (Pointer) haben, mit dem sie die Operation durchführen sollen.
- Eine Klasse ist nun ein Verbund der Daten eines struct-Types von c und der Funktionen dazu.
- Da Funktionen und Daten nun eine Symbiose bilden, kann der erste Parameter auf magische Weise entfallen.

Schritte vom c-Struct zur java-Klasse

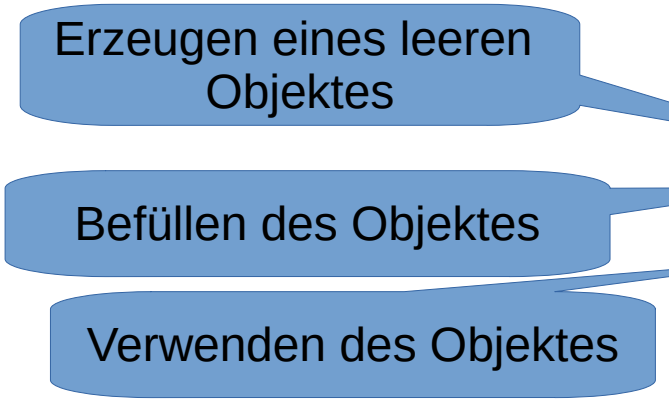
- Kopieren des Inhaltes von btime.h in Btime.java
- Kopieren des Inhalts von btime.c nach Btime.java
- Entfernen von typedef
- Schließende Klammer von stuct nach ganz unten
- Entfernen von btime nach ‚}‘
- Ersetzen von struct durch class
- Entfernen der Prototypen

Schritte vom c-Struct zur java-Klasse

- Entfernen der Pointersternchen
- -> ersetzen durch . -operator
- Malloc-zeile aus btimeCreate entfernen
- Entfernen des ersten Parameters (btime*) aus allen Funktionen
- In den Funktionen ersten Parameter ersetzen durch this.
- Akt. Datum ermitteln mittels Calendar aus java.util.

Schritte vom c-Struct zur java-Klasse

- Ergänzen einer main-Funktion
- Main kann in der selben Klasse (oder Datei) oder gesonderter Klasse (oder Datei) programmiert werden.




```
public static void main(String args[])
{
   Btime bt1=new Btime();
   bt1.btimeGetCurrentTime();
   bt1.btimeShow();
}
```

- Der erste Parameter der c-Funktionen ist vor den Funktionsaufruf getreten und taucht in der Funktion als this. Wieder auf.

Unsere erste Klasse

```
import java.util.*;

class Btime
{
 int h;
 int m;

 void btimeSetH(int h)
 {this.h=h;}

 void btimeSetM(int m)
 {this.m=m;}

 Btime btimeCreate()
 {
  h=m=0;
  return this;
 }

 void setCurrentTime()
 {
  Calendar c=Calendar.getInstance();
  this.h=c.get(Calendar.HOUR_OF_DAY);
  this.m=c.get(Calendar.MINUTE);
 }
```

```
 void btimeShow()
 {
  System.out.printf("%02d:%02d",
                          this.h,this.m);
 }

 void btimeIncrement()
 {
  m++;
  if (this.m==60)
   {this.m = 0; this.h++;}
  if (this.h==24)
   {this.h = this.m = 0;}
 }

 public static void main(String args[])
 {
  Btime bt1=new Btime();
  bt1.setCurrentTime();
  bt1.btimeShow();
 }
}
```

Main in gesonderter Klasse

```
class BTimeTest
{
    public static void main(String args[])
    {
      BTime t1=BTime.create();
      BTime t2= BTime.create();
      t1.setCurrentTime();
. . .
      System.out.println("");
      System.out.println("via toString. "+t1+" / "+t2);
    }
}
```

- Die Methode main kann auch in einer anderen Klasse programmiert werden.
- Beide Klassen müssen zunächst im selben Verzeichnis liegen

Anmerkungen zu this

- Hinter this verbirgt sich in den Memberfunktionen eine Referenz auf das Objekt, zu dem die Memberfunktion aufgerufen worden ist.
- Man kann this. beim Zugriff auf Member auch weglassen. In manchen Situationen braucht man this aber, z.B. bei Namensgleichheit von Funktionsparameter(n) und Member(n).

```
class abc
{
  int a;
  void setA(int a){this.a=a;}
  . . .
}
```

Klassen und Objekte

- Eine Klasse definiert einen benutzerdefinierten Datentyp.
- In der Regel stellt eine Klasse einen Verbund aus Daten (Membervariablen) und Funktionalität (Memberfunktionen/Methoden) dar.
- Instanzen einer Klasse bezeichnet man als Objekte.
- Objekte sind gekennzeichnet durch:
  - State (Status): Gesamtheit der Werte der Membervariablen
  - Behavior (Verhalten): Bestimmt durch die Gesamtheit der Memberfunktionen

Klassen und Objekte

- Methoden und Membervariablen bilden zusammen die Member einer Klasse.
- Nach außen kann die Sicht in eine Klasse eingeschränkt werden (information hiding).
- Es gibt public und private Member.
- private vor einer Variablen/Funktionsvereinbarung führt dazu, dass dieses Element der Klasse von außerhalb nicht zu sehen ist.
- public vor einer Variablen/Funktionsvereinbarung führt dazu, dass dieses Element von außen ohne Einschränkung sichtbar ist.
- Regel:
  - Daten sollten private sein,
  - Methode (Funktionen) können public sein.

Klassen und Objekte

- Klassen beschreiben Objekte oder können als Bildungsvorschrift für Objekte bezeichnet werden.
- Objekte werden grundsätzlich mit `new <Class_name>(…)` erzeugt.
- Variablen eines Klassendatentyps sind immer nur Referenzvariable (so etwas, wie ein Pointer in c, der zunächst NULL enthält).
- **`Btime bt1;`** ist eine Referenzvariable, vergleichbar mit einer Pointervariablen in c. Es existiert noch kein Objekt.
- Erst mit **`bt1=new Btime();`** wird ein Objekt erzeugt.

Initialisierung

- Zur Initialisierung von Objekten gibt es spezielle Funktionen – Constructoren.
- Ein Constructor trägt den Namen der Klasse als Funktionsname und hat keinen Returntyp.
- Ein Constructor der Klasse Btime könnte z.B. folgendes Aussehen haben:

```
Btime(){…}                    //in der Classdekaration
```

- Es kann mehrere Constructoren in einer Klasse geben. Sie müssen sich in ihrer Parameterliste unterscheiden. Man spricht von überladenen Constructoren.
- Die Daten eines Objektes sollen immer valide sein. Dafür haben der u.a. Constructoren zu sorgen.

Constructoren

```
class Btime
{
 int h;
 int m;

 void setH(int h)
 {this.h=h;}

 void setM(int m)
 {this.m=m;}

 Btime() // Constructor
 {
  Calendar c=Calendar.getInstance();
  this.h=c.get(Calendar.HOUR_OF_DAY);
  this.m=c.get(Calendar.MINUTE);
 }
 Btime(int h, int m) // Constructor
 {
  this.h=(h>=0 && h<24)?h:0;
  this.m=(m>=0 && m<60)?m:0;
 }
  . . .
 public static void main(String args[])
 {
      Btime bt1=new Btime();
      bt1.btimeShow();
 }
}
```

- Wir haben jetzt zwei überladene Constructoren
  - `Btime()`
  - `Btime(int h, int m)`
- Die Funktion createBtime kann nun entfallen.
- Die set-Funktionen (setter genannt) haben üblicherweise einen Namen, der sich aus ‚set‘, gefolgt vom Variablennamen zusammensetzt.

Constructoren

- Constructoren sind spezielle Funktionen zur Erzeugung von Objekten.
- Sie tragen als Funktionsnamen den Klassennamen und haben keinen Returntyp.
- Sie vereinen das Bereitstellen von Speicher mit der Initialisierung der Instzanzvariablen, mit dem Ziel, dass jedes existierende Objekt valide Daten enthält.
- Es kann mehre überladene Constructoren geben.

Constructoren

- Ein Constructor ohne Parameter heißt DefaultConstructor.
- Constructoren sollten public sein.
- Je nach angegebener Parameterliste wird der passende Constructor (so es ihn gibt – sonst Compilerfehler) aufgerufen.
- Hat eine Klasse keinen Constructor, so generiert der Compiler einen Defaultconstructor ohne jedwede Funktionalität. Die Instanzvariablen sind in diesem Falle mit 0 oder mit ihrem Initialisierungswert belegt!

Constructoren

- Ein Constructor wird automatisch bei der Erzeugung eines Objektes aufgerufen.
- Gibt es mehrere, überladene Constructoren, so wählt der Compiler auf Grundlage der angegebenen Parameter den passenden Constructor aus.

```
Btime bt1=new Btime();          // erster Constructor
Btime bt2=new Btime(9,19);      // zweiter Constructor
```

Memberfunktionen/Methoden

- Funktionen können auch überladen werden.
- Funktionen einer Klasse können sich gegenseitig durch this.func() aufrufen, wobei func hier für die aufzurufende Funktion steht und natürlich Parameter haben kann. Die Angabe von this. kann aber in der Regel auch entfallen.
- Funktionen, die nur den Wert einer Instanzvariablen setzen, heißen setter.
- Funktionen, die nur den Wert einer Instanzvariablen zurückgeben, heißen getter.

Memberfunktionen/Methoden

```
public void setH(int h)
   {this.h=h;}

public void setM(int m)
   {this.m=m;}
```

- Setter sollen dafür sorgen, dass die Instanzvariablen grundsätzlich nur valide Werte enthalten.
- Deshalb ändern wir hier ab!

```
public void setH(int h)
   {this.h=(h>=0 && h<24)? h:0;}

public void setM(int m)
   {this.m=(m>=0 && m<60)? m:0;}
```

Memberfunktionen/Methoden

- Memberfunktionen sollen so implementiert sein, dass die Daten eines Objektes immer valide Werte beinhalten.
- Memberfunktionen kann man funktional in Verwaltungsfunktionen, Implementierungsfunktionen, Hilfsfunktionen und Zugriffsfunktionen einteilen.
- Gültigkeitsbereiche in Memberfunktionen
  - lokaler Block, in dem der Bezeichner verwendet wird,
  - umfassende Blöcke innerhalb der Funktion, in der der Bezeichner verwendet wird,
  - Klasse, in der die Funktion als Memberfunktion deklariert worden ist,
  - wird ein Bezeichner in einem eingeschlossenen Gültigkeitsbereich erneut vereinbart, so verdeckt diese Vereinbarung die ursprüngliche Vereinbarung.

Die Methode toString

```
public String toString()
{
  return String.format("%02d:%02d", h, m);
}
```

- Die Funktion (Methode) toString bildet zu dem Objekt eine Stringrepräsentation – wandelt das Objekt in eine sinnvolle Zeichenkette um.
- Diese Zeichenkette kann dann mit + verkettet oder ausgegeben werden.
- Die Methode show() kann nun entfallen.
- In Ausgaben wird toString automatisch aufgerufen.

```
import java.util.*;

class BTime
{

 int h;
 int m;

  BTime() // Constructor 1
  {
   Calendar c=Calendar.getInstance();
   this.h=c.get(Calendar.HOUR_OF_DAY);
   this.m=c.get(Calendar.MINUTE);
  }
  BTime(int h, int m) // Constructor 2
  {
   this.h=(h>=0 && h<24)?h:0;
   this.m=(m>=0 && m<60)?m:0;
  }


  BTime (String s)  // Constructor 3
  {
    String a[]=s.split(":");
    if (a.length==2)
    {
      h=Integer.parseInt(a[0]);
      m=Integer.parseInt(a[1]);
    }else h=m=0;
  }
```

```
public void setH(int h)    // Setter
   {this.h=(h>=0 && h<24)? h:0;}



 @Override
 public String toString()
 {
   String s=String.format("%02d:%02d",h,m);
   return s;
 }

 void show()
 {
   System.out.printf("%02d:%02d",this.h,this.m);
 }
 void increment()
 {
   m++;
 if (m==60)
   {m = 0; h++;}
 if (h==24)
   {h = m = 0;}
 }
```

```
 public static void main(String args[])
  {
    BTime bt=new BTime();
    System.out.println(bt.toString());
    System.out.println();
    bt.increment();
    System.out.println(bt); // toString wird hier implizit aufgerufen!!
    System.out.println();
    // Verwendung von Kommandozeilenparametern
    // java BTime 9 36
    if (args.length==2)
    {
      BTime bx=new BTime(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
      System.out.println("bx: "+bx+" Uhr");
    }
  }
}
```

Experimentieren Sie mit diesem Beispiel

Nicht primitive Datentypen

- Die Datentypen boolean, char, die int- und Gleitpunktdatentypen nennt man in Java primitive Datentypen.
- Klassen und Arrays sind “nicht primitive“ Datentypen.
- Variablen von nicht primitiven Datentypen sind immer lediglich Referenzvariablen.
- Das eigentliche Datenobjekt wird gesondert erzeugt (vergl. in main, obiges Beispiel).

Referenzvariable / Objekt

- Im Beispiel der Klasse BTime gab es in main die Variable

  ```
  BTime bt;
  ```

- Dies ist eine Referenzvariable, Ein BTime Objekt existiert noch nicht.
- Das eigentliche Objekt wird erst mit

  ```
  bt=new BTime();
  ```

  erzeugt.
- Wie im Quelltext zu sehen kann beides zusammengefasst werden zu:

  ```
  BTime bt=new BTime();
  ```

Arrays - Referenzvariable

- Bei Arrays verhält sich die Sache ganz ähnlich
- Es wird zunächst eine Referenzvariable angelegt:
- `int array[];`
- `int []array;`
- Die Klammern können vor oder nach dem Namen des Arrays stehen, das ist egal.
- Achtung!! Die Klammern bleiben immer leer.

Arrays - length

- Alle Array in Java verfügen über eine Membervariable length.
- Sie enthält die Anzahl der Arrayelemente

```
class ArrayLength
{
 public static void main(String args[])throws Exception
  {
   //Array, durch Initialisierung erzeugt - ohne new
    int array[]={2,4,6,8,10,12};
    for (int i=0; i<array.length; i++)
    {
      System.out.printf("array[%d]: %d\n",i, array[i]);
    }
  }
}
```

Array – Erzeugen mit new

- Das eigentliche Array wird mit nun mit new angelegt.

```
int array[]=new int[20];
```

- Dabei entsteht ein Array von 20 int-Elementen mit noch undefiniertem Inhalt.
- Nun kann man beispielsweise mit einer Schleife das Array füllen:

```
for(int i=0; i<20;i++)array[i]=0;
```

Array – Erzeugen per Initialisierung

- Ein Array kann ebenfalls durch eine Initialisierung angelegt werden.

```
int array[]={1,3,5,7,9};
```

- Auch hier bleiben die eckigen Klammern leer.

Mehrdimensionale Arrays

- Erzeugung durch Initialisierung

```
int mda[][]={{1,2},{1,2,3},{1,2,3,4,5}};
```

- Erzeugung mit new-Operator

```
int mda[][];
mda=new int[2][10];
```

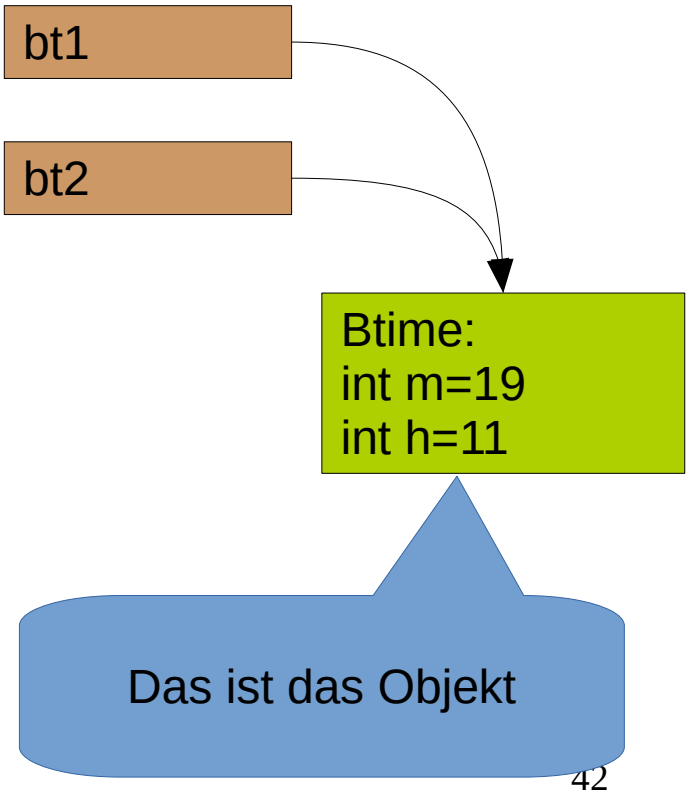
- oder

```
mda=new int[2][];
mda[0]=new int[2];
mda[1]=new int[5];
```

Zuweisungen von Daten nichtprimitiver Datentypen

- Bei der Zuweisung von Daten der nichtprimitiven Datentypen werden nur die Referenzen kopiert, nicht die Objekte.
- Nach Ausführung des Codes ergibt sich die Konstellation auf dem Bild.
- Genauso verhält es sich bei Parameterübergabe an Funktionen.
- Alle Änderungen die am Objekt über bt1 vorgenommen werden, ändern sich auch für bt2, es gibt ja nur das eine Objekt.

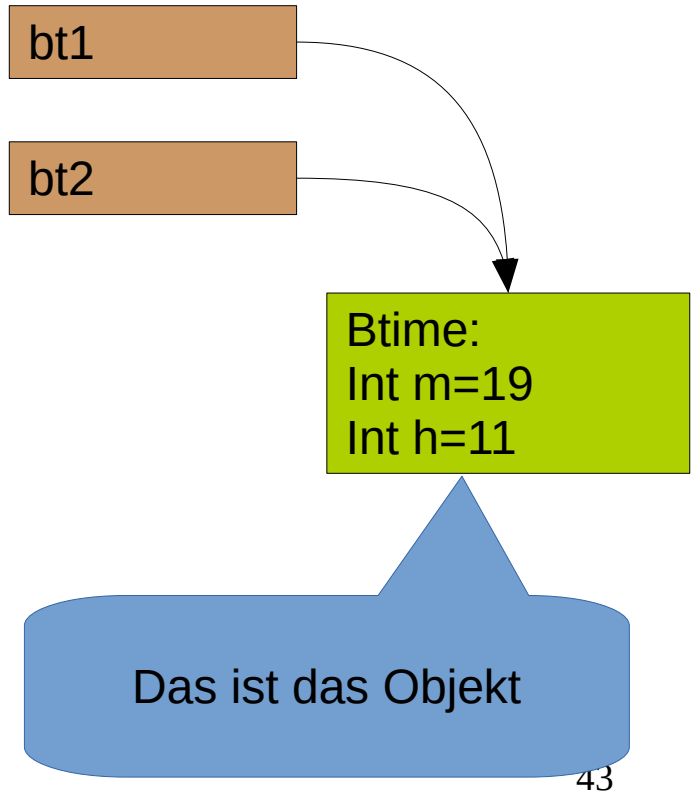
```
BTime bt1=new Btime();
BTime bt2=bt1;
```



Vergleich von Daten nichtprimitiver Datentypen

- Bei dem Vergleich auf Gleichheit (== oder !=) von Daten nichtprimitiver Datentypen wird geprüft, ob es sich um ein und dasselbe Objekt handelt.
- Es wird nicht geprüft, ob zwei Objekte sich gleichen.
- Im Beispiel wird true erzeugt.

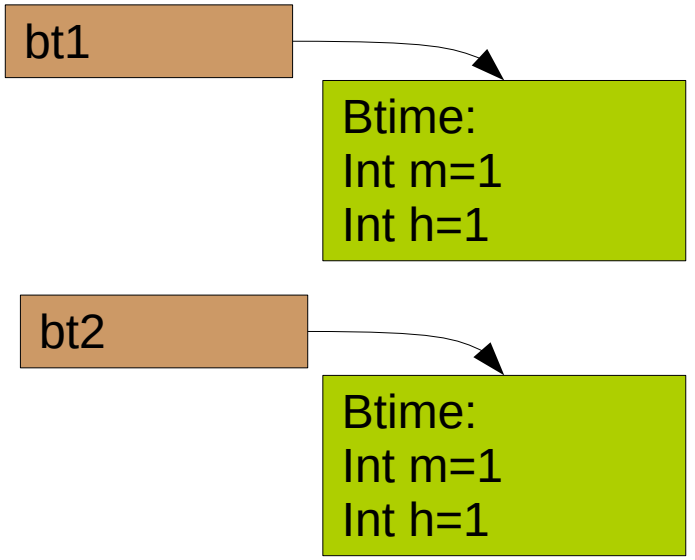
```
if (bt1==bt2)
```



Vergleich von Daten nichtprimitiver Datentypen

- Bei diesem Beispiel gibt es zwei Objekte bt1 und bt1
- Beide Objekte haben die Werte 1 für Minute und Stunde, sie gleichen einander.
- Der Vergleich `if(bt1==bt2)` liefert aber false, weil es zwei Objekte sind.

```
Btime bt1=new BTime(1,1);
Btime bt2=new BTime(1,1);
. . .
if (bt1==bt2)
```



Vergleich von Daten nichtprimitiver Datentypen

- Sollen zwei Objekte auf gleichen Inhalt getestet werden, so ist die Funktion equals zu verwenden.
- Die Anwendung von

  `if(bt1.equals(bt2))`

  liefert true, weil sich die Objekte vollständig gleichen.

```
Btime bt1=new BTime(1,1);
Btime bt2=new BTime(1,1);
. . .
if (bt1.equals(bt2))
```