Abstrakte ***D***aten***T***ypen (ADT)

- Dynamische Datenstrukturen
  - Beispiel Lineare Liste
  - Motivation ADT
- Abstrakte Datentypen
  - Beispiele
  - Definition ADT
  - Definition Signatur
- ADT *Punkt*
  - Implementierung in Java

Motivation

- Arrays (in fast allen Programmiersprachen) haben den gleichen Nachteil:
  Man kann ihre Länge nicht ändern
- Das ist eine Konsequenz aus der Speicherverwaltung!
  - Mehr Flexibilität -> weniger Performance

- Alternative: dynamische Datenstrukturen
- Beispiele
  - (lineare) Liste
  - (binärer) Baum
  - Graph
  - Stack (Stapel, LIFO)
  - Queue (Schlange, FIFO)
  - Menge

Beispiel: add in der Klasse BigInt mit Arrays

```
void add(BigIntegerWithArray no2) {
    int maxLength;
    if (this.digits.length > no2.digits.length)
        maxLength = this.digits.length;
    else
        maxLength = no2.digits.length;
    int[] sum = new int[maxLength];
    int carryOver = 0;
    int i1 = this.digits.length - 1, i2 = no2.digits.length - 1, sumI = maxLength - 1;
    // Stellen in beiden Zahlen
    while (i1 >= 0  && i2 >= 0 ) {
        sum[sumI] = (this.digits[i1] + no2.digits[i2] + carryOver) % 10;
        carryOver = (this.digits[i1] + no2.digits[i2] + carryOver) / 10;
        i1--;
        i2--;
        sumI--;
    }
    // Zahl "this" ist länger
    while (i1 >= 0) {
        sum[sumI] = (this.digits[i1] + carryOver) % 10;
        carryOver = (this.digits[i1] + carryOver) / 10;
        i1--;
        sumI--;
    }
    // Zahl "no2" ist länger
    while (i2 >= 0) {
        sum[sumI] = (no2.digits[i2] + carryOver) % 10;
        carryOver = (no2.digits[i2] + carryOver) / 10;
        i2--;
        sumI--;
    }
    if (carryOver > 0) {
        int[] extendedArray = new int[sum.length + 1];
        for (int i = 0; i < sum.length; i++)
            extendedArray[i + 1] = sum[i];
        sum = extendedArray;
        sum[0] = carryOver;
    }
    digits = sum;
}
```

Die markierten Teile werden nur benötigt, weil man mit einer festen Array-Länge umgehen muss!

Speichert man die Ziffern in einer *Liste*, die dynamisch nach Bedarf wachsen kann entfallen alle diese Programmteile!

Mögliche Operationen, die eine solche Liste zur Verfügung stellt 1/2

- addFirst: Liste × Element → Liste
  - fügt ein neues Element am Beginn einer Liste ein
- addLast: Liste × Element → Liste
  - hängt ein neues Element am Ende einer Liste an
- getFirst: Liste → Element
  - liefert das erste Element der Liste
  - Vorbedingung: Liste ist nicht leer
- getLast: Liste → Element
  - liefert das letzte Element der Liste
  - Vorbedingung: Liste ist nicht leer
- getAt: Liste × Position → Element
  - liefert das Element an der angegebenen Position der Liste
  - Vorbedingung: Liste hat mindestens so viele Elemente
- removeFirst: Liste → Liste
  - löscht das erste Element der Liste
  - Vorbedingung: Liste ist nicht leer

Mögliche Operationen, die eine solche Liste zur Verfügung stellt 2/2

- contains: Liste × Element → boolean
  - sucht ein Element in einer Liste; Ergebnis = *true*, wenn Element in der Liste, sonst *false*
- delete: Liste × Element → Liste
  - entfernt ein Element aus der Liste, sofern es drin ist
- clear: Liste → Liste
  - entfernt alle Elemente aus der Liste
- isEmpty: Liste → boolean
  - liefert *true* genau dann, wenn die Liste leer ist, sonst *false*
- size: Liste → int
  - liefert die Länge der Liste, d.h. die Anzahl der Elemente
- clone: Liste → Liste
  - liefert eine (flache) Kopie der Liste

- empty: → Liste
  - erzeugt eine neue leere Liste

```
void add(BigIntegerWithListV3 no2) {
    List sum = new List();
    int carryOver = 0;
    int i1 = this.list.size() - 1, i2 = no2.list.size() - 1;
    // Stellen in beiden Zahlen
    while (i1 >= 0  && i2 >= 0 ) {
        sum.addFirst((this.list.getAt(i1) + no2.list.getAt(i2) + carryOver) % 10);
        carryOver = (this.list.getAt(i1) + no2.list.getAt(i2) + carryOver) / 10;
        i1--;
        i2--;
    }
    // Zahl "this" ist länger
    while (i1 >= 0) {
        sum.addFirst((this.list.getAt(i1) + carryOver) % 10);
        carryOver = (this.list.getAt(i1) + carryOver) / 10;
        i1--;
    }
    // Zahl "no2" ist länger
    while (i2 >= 0) {
        sum.addFirst((no2.list.getAt(i2) + carryOver) % 10);
        carryOver = (no2.list.getAt(i2) + carryOver) / 10;
        i2--;
    }
    if (carryOver > 0)
        sum.addFirst(carryOver);
    list = sum;
}
```

BigInt-Implementierung als Liste (Annahme: die Elemente sind vom Typ int)

-> viel einfacher / kürzer!

Offene Fragen und ihre Konsequenzen

1. Wie genau funktionieren die Listen „intern“?
2. Was genau passiert, wenn...

Als Benutzer einer solchen Liste (zum Beispiel für BigInt) ...

- ... ist uns 1. egal, aber
- ... müssen wir 2. genau wissen

Konsequenzen

1. -> „Liste“ ist ein Datentyp, von dessen „interner“ Implementierung wir abstrahieren können:
   Abstrakter Datentyp = ADT
2. -> Wir brauchen eine genaue (formale) Spezifikation

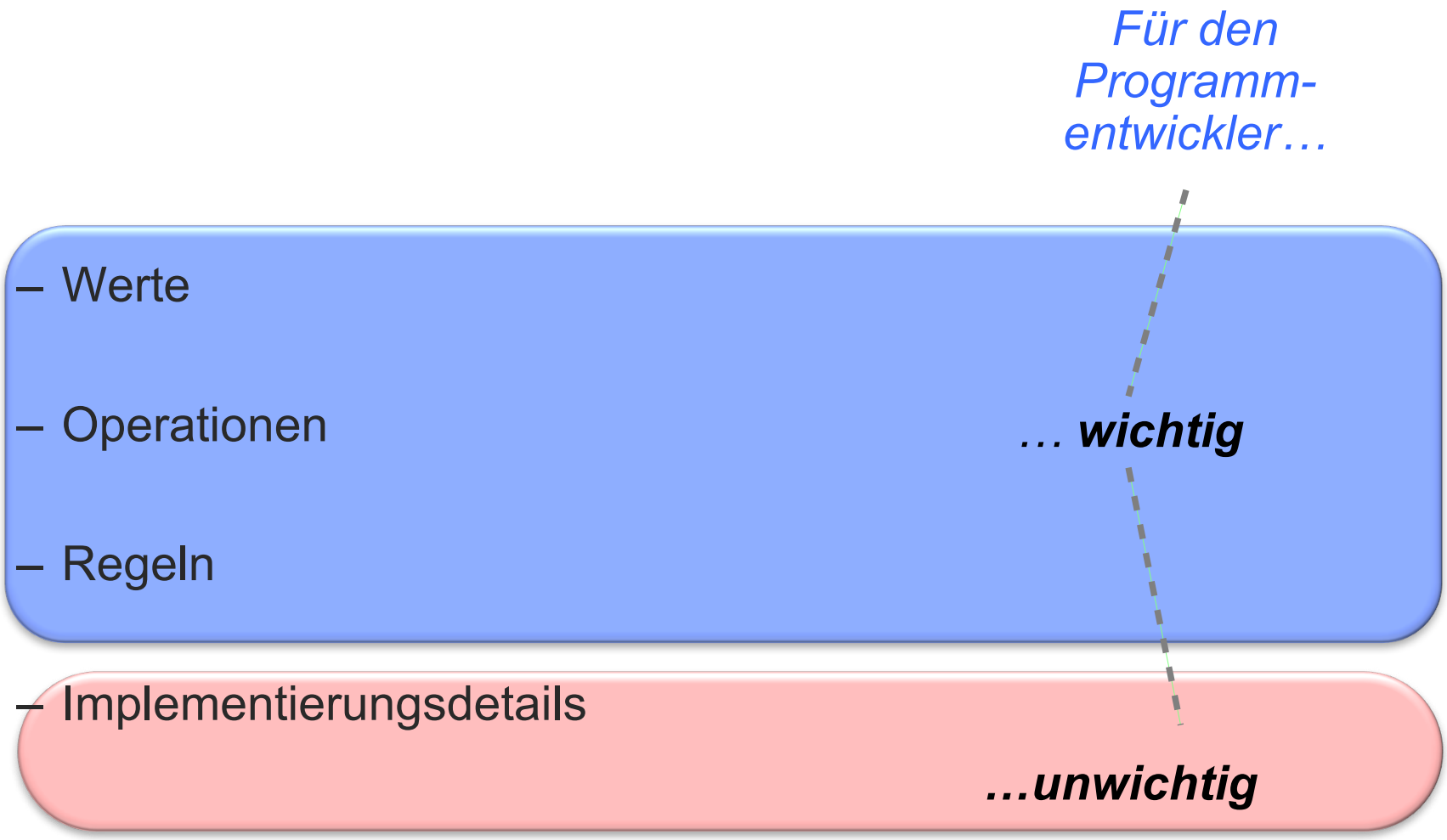
Bekannte Datentypen aus Java: elementare

- byte, short, int, long, char,
  float, double
  - Wir wissen, welche Operationen (rechnen, vergleichen) es auf Zahlen gibt
- boolean
  - Wir wissen, welche Operationen es auf Booleans gibt
- Von elementaren Datentypen kennen wir viel von ihrer internen Darstellung und Arbeitsweise
  -> Erklärung für int-Overflow etc.

Bekannte Datentypen aus Java: abstrakte

- Array
  - Wir wissen wenig davon, wie Arrays intern funktionieren
  - Wir wissen aber, wie ihre Indizierung und die Zuweisung funktionieren -> das reicht!
- String
  - Wir wissen nicht, wie Strings intern funktionieren
  - Es gibt bestimmte Operationen auf Strings, die wir benutzen können ohne zu wissen, *wie* genau sie funktionieren
  - Wir wissen aber, *was* sie tun -> das reicht!
- BigInt
  - Es ist nicht nötig, den internen Aufbau zu kennen, um z.B. zwei sehr große Zahlen zu addieren:
    BigInt zahl1 = new BigInt(“123456789012345678 90");
    BigInt zahl2 = new BigInt("456789012345678901 23");
    zahl1.add(zahl2);
  - Weil wir die Addition kennen, ist eine Beschreibung von „add“ nicht nötig
  - Zugegeben: Die Klasse kann (noch) nicht so viel...

Datentyp *int*



Datentyp *int*: wichtige Informationen

- Werte, Datenbereich

  1, 2, -54, …

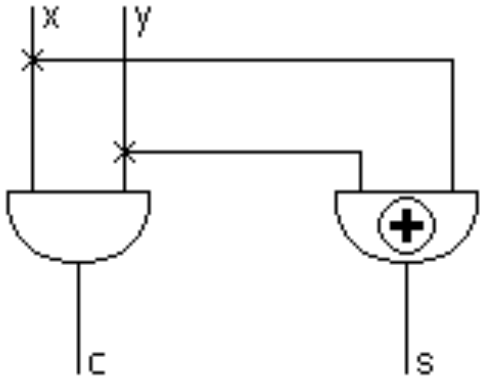
- Operationen auf *int*

  +, -, *, /, ==, >, <, ...

- Regeln
  - Punkt-vor-Strich
  - Assoziativ-, Kommutativ-, Distributivgesetz
  - Transitivität

Datentyp *int*: unwichtige Implementierungsdetails

- Implementierung (Auswahl)
  - Interne Darstellung
    - → Binär 100101001
    - → 1. Zeichen Vorzeichen
  - Addition
    - → Halbaddierer



  - Multiplikation mit 2
    - → shift nach links
      0101 * 2 → 1010

Datentyp *Menge*: wichtige Informationen

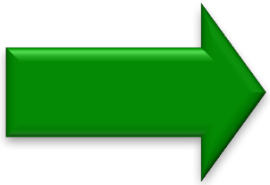
- Werte, Datenbereich:
  - Zahlen Menge von Zahlen
  - 0, 1, 2, … { }, {1, 3, 5}, …
- Operationen auf der Menge
  - erzeugen, einfügen, wegnehmen, schauen-ob-leer, …
- Regeln
  - Eine leere Menge wird erzeugt → Menge ist leer.
  - Eine Zahl wird in Menge eingefügt → Menge ist nicht leer.
  - Menge ist leer,
    eine Zahl wird eingefügt,
    diese Zahl wird weggenommen
    → Menge leer.

Datentyp *Menge*: unwichtige Implementierungsdetails

- Implementierung

Für den Benutzer von Datentypen wichtig …

…ist, *was* sie tun (= Schnittstelle)
nicht, wie sie es tun (= Implementierung)



Datentypen abstrahieren

- Abstraktion
  - Datentypen *abstrahieren* von der Implementierung
  - Prinzip wird in der Informatik häufig verwendet:
    - → „Bit“ *abstrahiert* von physikalischen Zuständen
    - → CPU *abstrahiert* von einer Menge von Bits und deren Zustandsänderungen
    - → Java Virtual Machine *abstrahiert* von konkreter CPU

Definition: Abstrakter Datentyp

- Spezifikation
  - Ein ***Abstrakter Datentyp*** beschreibt
    - → eine *Menge von Werten* und
    - → *Operationen* auf diesen Werten,
      die *nur* über eine *Schnittstelle* zugänglich sind, sowie
    - → *Regeln* über die Wirkung der Operationen auf den Werten und
    - → ist *gekapselt*.

Verhalten (Semantik) aus Benutzersicht

- Implementierung (eines ADT)
  - Ein Programm(teil), das den Datentyp realisiert.

Anmerkung

- Zur Implementierung eines ADT werden typischerweise Datenstrukturen aus vorhandenen Datentypen gebildet, z.B.
  - → Arrays haben einen Elementtyp
  - → Strings bestehen aus Zeichen
  - → Klassen haben Felder aus (einfachen) Typen, z.B.
    Wohnadressen bestehen aus Straßen- und Ortsangabe (String), Postleitzahl und Hausnummer (int)

Definition: Signatur einer Operation

- Eine ***Signatur*** ist eine Liste mit
  - dem Namen,
  - den Typen der Eingabewerte, und
  - dem Typ des Rückgabewertes

  einer Operation.

Achtung:

- In Java gehört der Typ des Rückgabewertes *nicht* zur Signatur!
- Methoden, deren Kopf sich nur durch den Typ des Rückgabewertes unterscheidet, sind nicht erlaubt

ADT *Point*: Punkte im kartesischen Koordinatensystem

- Typ: Point
- Operationen:

create	double × double → Point
xValue	Point → double
yValue	Point → double
isOrigin	Point → boolean
distance	Point × Point → double
translate	Point × double × double → Point

- Regeln:

xValue(create(x, y)) = x
yValue(create(x, y)) = y
isOrigin(create(x, y)) = true ⇔ x = 0 ∧ y = 0
translate(create (x, y), a, b) = create(x+a, y+b)
...

ADT *Point*: Punkte im kartesischen Koordinatensystem

- Typ: Point
- Operationen:

create	double × double → Point
xValue	Point → double
yValue	Point → double
isOrigin	Point → boolean
distance	Point × Point → double
translate	Point × double × double → Point

Eigentlich müsste man $\mathbb{R}$ schreiben statt *double*

- Regeln:

xValue(create(x, y)) = x
yValue(create(x, y)) = y
isOrigin(create(x, y)) = true ⇔ x = 0 ∧ y = 0
translate(create (x, y), a, b) = create(x+a, y+b)
...

Idee: Zusammenspiel *aller* Funktionen muss beschrieben sein

Das sind die Signaturen dieser Operationen

ADT *List*: Liste von ganzen Zahlen

- Typ: List
- Operationen:

create	$\rightarrow$ List
addFirst	List $\times$ int $\rightarrow$ List
removeFirst	List $\rightarrow$ List
getFirst	List $\rightarrow$ int
getAt	List $\times$ int $\rightarrow$ int
isEmpty	List $\rightarrow$ boolean
length	List $\rightarrow$ int
clone	List $\rightarrow$ List

- Regeln:

isEmpty(create()) = true
getFirst(addFirst(L, a)) = a
getAt(addFirst(L, a), 0) = a
isEmpty(removeFirst(addFirst(create(), e))) = true
length(addFirst(L, e)) = length(L)+1
...

ADT *Set*: Menge von Zeichen

- Typ: Set
- Operationen:

create	$\rightarrow$ Set
add	Set $\times$ Character $\rightarrow$ Set
remove	Set $\times$ Character $\rightarrow$ Set
isEmpty	Set $\rightarrow$ boolean
card	Set $\rightarrow$ int
contains	Set $\times$ Character $\rightarrow$ boolean
clone	Set $\rightarrow$ Set

- Regeln:

isEmpty(create()) = true
remove(add(S, e), e) = S
isEmpty(remove(add(create(), e), e)) = true
contains(S,e) = true $\Longrightarrow$ card(S) = card(add(S, e))
contains(S,e) = false $\Longrightarrow$ card(S) = card(add(S, e)) – 1
...

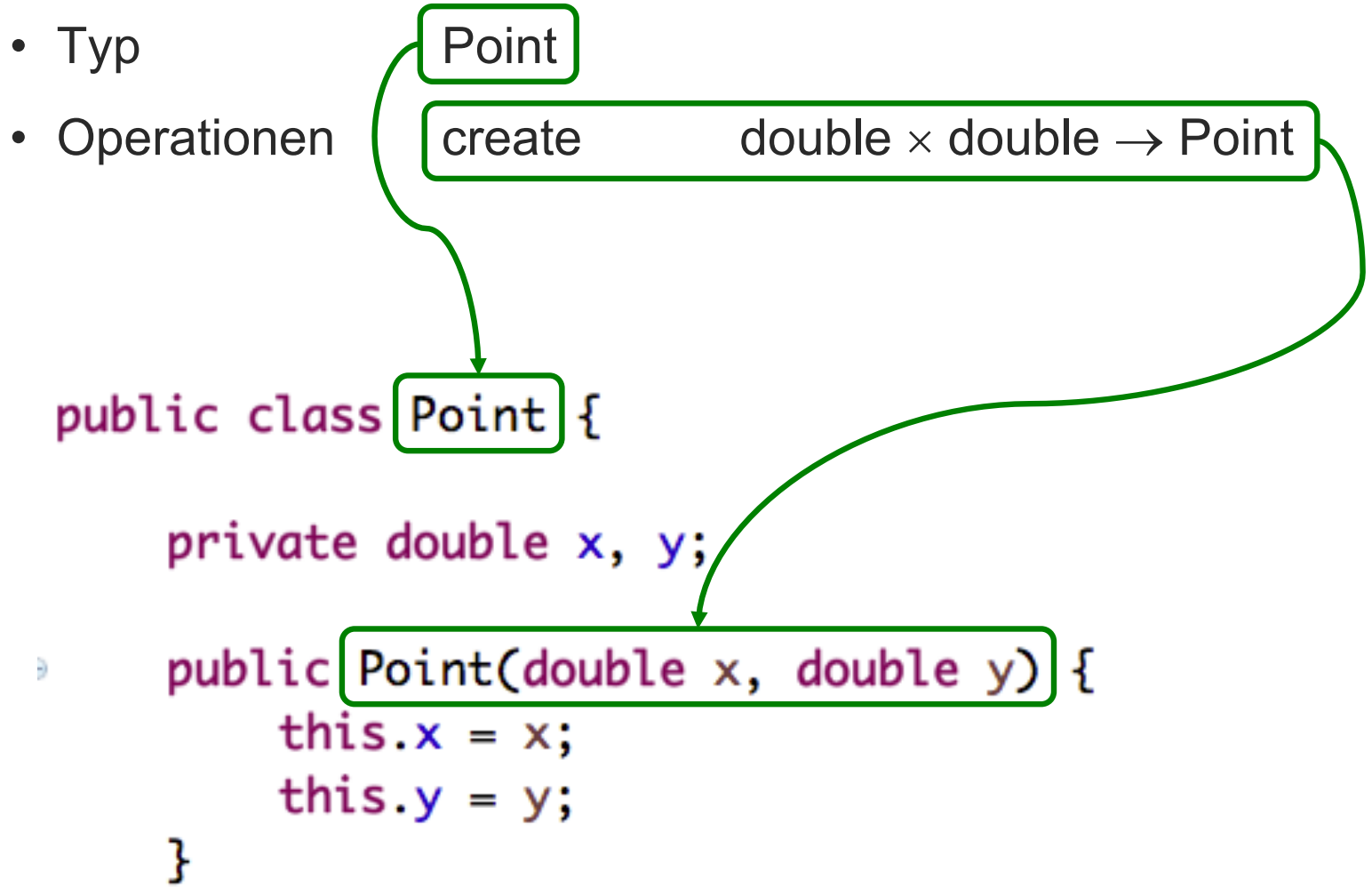
ADT in Java

Wir implementieren ADT in Java

- *Typ* (= Klassenname) muss angeboten werden;
  → für Benutzer nutz- (und damit sicht-)bar: public
- *Operationen* müssen angeboten werden;
  → für Benutzer nutz- (und damit sicht-)bar: public
- *Werte* sind gekapselt: private
- *Regeln* müssen eingehalten werden
  → für Benutzer erfahrbar, in Programmen, in denen der ADT benutzt wird und in Testprogrammen

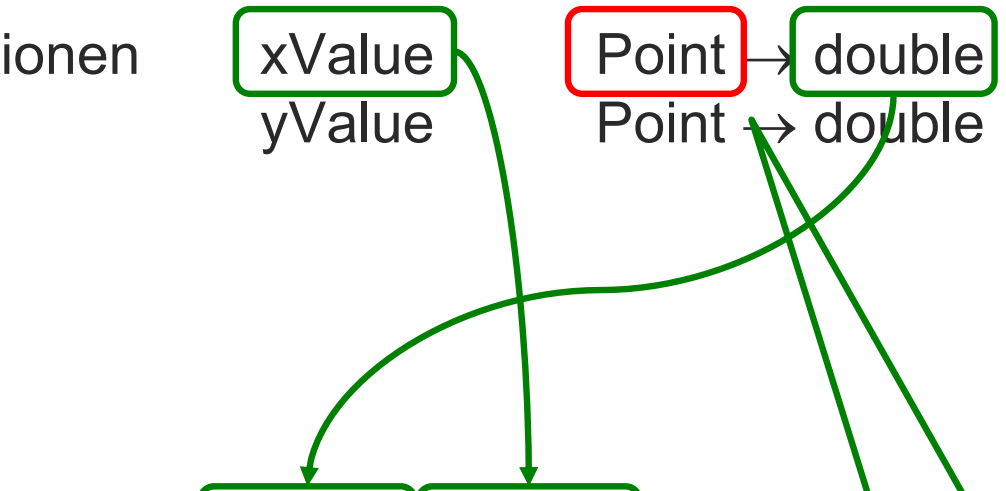
Umsetzung von ADT *Point* in eine Java-Klasse 1

- Typ
- Operationen



Umsetzung von ADT *Point* in eine Java-Klasse 2

- Operationen




```
public double xValue() {
    return x;
}

public double yValue() {
    return y;
}
```

Beim Aufruf wenden wir uns an ein Objekt vom Typ *Point*; die Methode steht in der Klasse *Point*

Umsetzung von ADT *Point* in eine Java-Klasse 3

- Operationen
  - isOrigin: Point $\rightarrow$ boolean
  - distance: Point $\times$ Point $\rightarrow$ double

```
public boolean isOrigin() {
    return x == 0 && y == 0;
}

double distance(Point other) {
    return Math.sqrt((x - other.x)*(x - other.x) + (y - other.y)*(y - other.y));
}
```

Umsetzung von ADT *Point* in eine Java-Klasse 4

- Operationen translate Point × double × double → Point

- Variante 1: der Punkt selbst wird verändert

```
public void translate(double dx, double dy) {
    x += dx;
    y += dy;
}
```

Diese Variante ist üblich

- Variante 2: der Punkt bleibt unverändert

```
public Point translate(double dx, double dy) {
    return new Point(x + dx, y + dy);
}
```

Die Signatur würde aber auch Variante 2 erlauben...

Umsetzung von ADT *Point* in eine Java-Klasse 5

- Regeln

  xValue(create(x, y)) = x
  yValue(create(x, y)) = y
  isOrigin(create(x, y)) = true $\Leftrightarrow$ x = 0 $\wedge$ y = 0
  translate(create (x, y), a, b) = create(x+a, y+b)

- Variante 1: (beliebige) Testwerte in einer *main*-Methode

```
public static void main(String[] args) {

    println(new Point(24, 26).xValue() == 24);
    println(new Point(24, 26).yValue() == 26);
    println(new Point(24, 26).isOrigin() != true);
    println(new Point(0, 0).isOrigin() == true);

    Point translated = new Point(24, 26);
    translated.translate(5, 7);
    println(new Point(24+5, 26+7).equals(translated));
}
```

Umsetzung von ADT *Point* in eine Java-Klasse 6

- Regeln

  xValue(create(x, y)) = x
  yValue(create(x, y)) = y
  isOrigin(create(x, y)) = true $\Leftrightarrow$ x = 0 $\wedge$ y = 0
  translate(create (x, y), a, b) = create(x+a, y+b)

- Variante 2: (beliebige) Testwerte in einer JUnit-Test-Methode

```
@Test
public void axiomExamples() throws Exception {

    assertEquals(24, new Point(24, 26).xValue(), 0d);
    assertEquals(26, new Point(24, 26).yValue(), 0d);
    assertFalse(new Point(24, 26).isOrigin());
    assertTrue(new Point(0, 0).isOrigin());
    Point translated = new Point(24, 26);
    translated.translate(5, 7);
    assertEquals(new Point(24+5, 26+7), translated);
}
```

Zusammenfassung

- Abstrakte Datentypen sind von zentraler Bedeutung für die Realisierung komplexer Strukturen
- Ein ADT
  - hat eine *Menge von Werten*
  - hat *Operationen* auf diesen Werten
    die nur über eine *Schnittstelle* zugänglich sind,
  - definiert *Regeln* über die Wirkung der Operationen auf den Werten
  - ist *gekapselt* (hat eine verborgene Implementierung).
- Das Konzept der ADT lässt sich durch objektorientierte Programmiersprachen direkt umsetzen
- Anmerkung:
  Wir werden Java-Interfaces für die ADT-Spezifikation nutzen